USENIX

UNIX SECURITY II WORKSHOP

SUMMER 1990

U

US

USE

U NIX

USENIX

WORKSHOP PROCEEDINGS

UNIX SECURITY II

August 27-28, 1990
Portland, Oregon

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $13 for members and $16 for nonmembers.

Outside the U.S.A and Canada, please add
$7 per copy for postage (via air printed matter).

Past USENIX Security Workshop Proceedings

UNIX Security Workshop	1988	Portland, OR	$7

Outside the U.S.A and Canada, please add
$5 per copy for postage (via air printed matter).

Program and Table of Contents

UNIX Security Workshop II
Portland, OR
August 27-28, 1990

Monday, August 27

Tuesday, August 28

Program Chair:

Matt Bishop
Dept. of Mathematics and Computer Science
Bradley Hall
Dartmouth College
Hanover, NH 03755

The MITRE User Authentication System

David S. Goldberg
The MITRE Corporation
Bedford, MA

Introduction

To enhance the security of its unclassified computer facilities, MITRE developed a user authentication system to be used in addition to the standard login and password authentication provided by the individual systems themselves. The user authentication system is used to protect MITRE computers accessible via dial-in lines and via connections from the Internet. Dial-in connections are to a MICOM switch — no individual hosts are allowed to accept dial-ins — and only a limited number of hosts are permitted to accept Internet connections. All external connections are run through the authentication process. Local connections are not given the extra protection.

Dial-in Protection

The Dial-in protection part of the user authentication system has three components:

1. A smartcard that provides users with a unique "passcode" each time they connect.
2. A host designated to maintain the smartcard database and run the software that authenticates the user's passcode.
3. A MICOM data switch, to which all dial-ins must connect.

The smartcard used at MITRE is the SecurID, sold by Security Dynamics, Inc. The card has an LED window which displays a 6 digit Pseudo Random Number (PRN) generated by a proprietary algorithm which takes as input a seed programmed into the card by the vendor, and the current time.

The host that maintains the smartcard software is a DEC VAX 8200 running ULTRIX. The SecurID database stores user information, the card serial number and the PRN seed. In addition, it generates the user's Personal Identification Number (PIN), which the user uses to identify the card when he/she logs in. The software and the card run the same PRN algorithm, so the software can determine the number that should be displayed on the card. A user shows that he/she has the card and knows the PIN by typing in the correct passcode (PIN followed by PRN). The software compares the PRN it generates with the one entered by the user. If they match, the user is considered authentic.

The MICOM switch is configured to route connections from dial-in modems to one of a specific set of tty ports on the 8200. A version of the getty program, modified to skip the login prompt, is used to run a program called acmlgn[1]. Acmlgn prompts the user for a passcode. The user's input is compared to the PIN and PRN retrieved from the database software and if they match, the user is passed back to the MICOM, where he can choose any of the available services.

[1] The acmlgn command is part of the code MITRE received from Security Dynamics. MITRE modified it slightly to work with the MICOM switch

A typical session looks like the following:

```
User dials appropriate phone number, sets modem and sees:

Connected to channel xx/yyy

Enter Passcode: ***********

Access Denied


Enter Passcode: **********

Enter MITRE System or RETURN for a Menu:

:
```

If the user enters the passcode incorrectly, or any other error occurs, the Access Denied message is printed. In the above example, 11 characters were typed instead of the requisite 10. The PIN and PRN are concealed by the acmlgn program. It turns off echo on the terminal line and prints an asterisk for each character typed. It also handles delete and backspace characters properly. If the user inputs 3 bad passcodes, the program exits, disconnecting the user from the system. In addition to authentication, the passcode procedure can be used to initialize a PIN for a new card and synchronize the card's internal clock with the host.

Internet Protection

Protection from attacks on Internet connected hosts is accomplished by a similar mechanism. Rlogin, telnet and ftp sessions from hosts determined to be outside the MITRE perimeter are challenged for a passcode before receiving a login prompt. The system is built on three components:

1. A filtering Internet gateway that limits the number of systems exposed to external communications.
2. The SecurID smartcard system.
3. A user authentication server and its associated clients.

MITRE uses a filtering gateway from cisco Systems, Inc, to limit Internet connections to a small number of hosts that run the SecurID client software. The gateway simply drops packets destined for other hosts that are on our internal network, but are not allowed to communicate with non-MITRE hosts.
Those hosts that are allowed to communicate with the Internet are given additional protection in the form of the SecurID smartcard system. We have implemented a network service to handle the SecurID system. Like most UNIX network services, our SecurID service is implemented by the server/client model of communication. Currently the only systems for which these clients have been developed are UNIX systems.

The UA server runs on the same ULTRIX 8200 as the acmlgn program. It listens on a well known port (we chose 501) and waits for connections. The client sends the passcode entered by the user to the server, which checks it and returns either a success or failure message to the client. It also appropriately handles new PIN requests and clock synchronization. The server may be used by any application that uses the appropriate protocol. We have concentrated on interactive session programs (TELNET, RLOGIN and FTP), but a client can be implemented for any other program that a site wants to protect.

The UA client programs we developed are used by other Internet services. For the case of TELNET and RLOGIN it was simplest to modify the standard (BSD based) UNIX login program. When the login program detects a telnet session (it is called with the -h flag by the telnetd daemon), it forks a call to the ualogin program. Ualogin prompts the user for passcode in much the same manner as acmlgn does for dial-in connections. It reads in the user's response and sends it to the server. If it gets three failures back from the server, it exits with a non-zero status, which tells the login program to exit the telnet connection. The rlogin program is handled similarly, although the fork is done after allowing the RLOGIN protocol to send over the remote environment. This was necessary because the RLOGIN protocol uses standard input to send this data, and ualogin uses standard input to interact with the user. From the user's point of view there is no difference: the prompt for passcode comes before the prompt for password.

The UA client for the FTP protocol works differently due to the nature of the ftp program. Because ftp does not open a pty, there is no way to call a program like ualogin. Also, there is no provision for an interactive sub program in the FTP protocol. To alleviate this, we developed the ua_ftp program. Ftp calls ua_ftp twice - once to send the server the remote host's IP address to determine whether the host is considered inside or outside the MITRE perimeter and, if it is not a MITRE host, a second time to send it the user's passcode. The user input is taken by the FTP protocol's ACCT command. The passcode and IP address are passed to ua_ftp via command line argument. One problem with this is that many, if not most, ftp client programs do not turn off echo when prompting for ACCT, which means that the user's passcode is displayed on the screen in clear text. Currently, we have decided that telling our users to clear the screen as soon as possible after using ftp is our only solution to this problem. The problem is not too serious because even if someone sees the PIN, they must also get the card before they can successfully use it.

The UA Protocol

The UA server and client programs communicate according to the UA protocol which includes the following commands:

Server	Client
READY	START
SUCC nnnnn	CLOSE
ERROR	NUMBER nnnnnnnnnn
NEWPIN	NUMBER nnnnnnnnnnnn
PINNUM nnnn	PINSEND
SYNCH	PINSTOP
APPROVED	NEXT nnnnnn
	RHOST nnn.nnn.nnn.nnn

The READY, START and CLOSE commands handle the administration of the connection. The server sends READY when the socket connection is completed. If the client ever gets anything it doesn't understand (noise, for example), it sends a START to restart the communication from the beginning – the server responds with READY. When the client has either determined that the user has successfully logged in or completely failed, it sends a CLOSE and exits.

The RHOST and APPROVED functions are used to allow the client to ask if an incoming connection has to be checked for passcode. The server takes the IP address argument from the RHOST command, checks it against four lists that are used to determine whether a host is allowed (no passcode necessary) or disallowed. The default is to disallow. The first contains hosts that are explicitly disallowed, the second, hosts that are explicitly allowed. The third and fourth lists contain network numbers (the first three fields of the IP address). The third contains those networks that are disallowed, the fourth, those that are allowed. If the host is to be allowed, the server sends the APPROVED message. Otherwise it sends an ERROR.

The two forms of the NUMBER command are used for sending either a PIN-PRN combination or the card's serial number along with the PRN, which indicates that the user is receiving his/her PIN for the first time. This can only happen when the card is initialized or if the system administrator resets the PIN. An ERROR message is returned to the client if the PIN-PRN combination does not match or the user tries the serial number-PRN combination after the card has already been initialized. If the new PIN request is genuine, the server sends the NEWPIN message to the client, which then asks the user whether he/she wants to receive the PIN. If the user responds in the affirmative, a PINSEND is sent to the server which in turn responds with PINNUM followed by the PIN. The client then displays the new PIN to the user. If the user decides not to receive the PIN, the client sends a PINSTOP to the server, which responds with ERROR. The client then asks the user to again enter passcode, or exits if there have been three tries. This works under both the dial-in and TELNET/RLOGIN clients. The FTP protocol does not allow for this interaction. Therefore, a NEWPIN from the server prompts the ua_ftp program to tell the user to try telnet and exit.

When a PIN-PRN combination is received, the server checks not only what the current PRN should be, but also if it would have been correct in the past minute, or will be correct the next. If either of these conditions hold, the server sends a SYNCH to the client. The user is then prompted for the next PRN to display on his/her card. If that matches what the server thinks it should be, it synchronizes its database with the clock on the card and returns SUCC. If not, it returns ERROR. Again, the ua_ftp program cannot handle the SYNCH in this manner, so it simply informs the user of the problem and tells him/her to use telnet to resolve it.

If the PIN-PRN combination match, the server sends a SUCC message. The client then returns CLOSE and exits with appropriate status. The SUCC command has an argument, which can contain any string. Currently we use the employee number, and have the client log it, which makes it easier to trace what card was used should we need to investigate a connection.

Conclusion and Future Work

We have been running the dial-in phase of the SecurID system for almost three years and the Internet phase for about 18 months. We have been very pleased with the system overall. During one of the recent hacker scares, we discovered many connection attempts that appeared to be from a computer running a program that tried many common login names. None of these got through.

There is much work to be done, however. The current implementation of the Internet server is not very portable, and is not well structured. An effort is underway to write the protocol as a yacc grammar which will provide better maintainability and portability. It will also help remove most, if not all, of Security Dynamics' code except for the database management routines themselves. This would allow us to link new database libraries without having to rewrite the code entirely unless, of course, the subroutine names and arguments are all changed.

The code is currently not available to the general public. There is a significant amount of Security Dynamics proprietary source in it. The dial-in system is almost entirely written by Security Dynamics, and the Internet system was built from that same code.

The UA protocol is currently tailored to the requirements of the SecurID device. It could, however, be generalized to work with any other such system.

"Foiling the Cracker": A Survey of, and Improvements to, Password Security[†]

Daniel V. Klein

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15217
dvk@sei.cmu.edu
+1 412 268 7791

ABSTRACT

With the rapid burgeoning of national and international networks, the question of system security has become one of growing importance. High speed inter-machine communication and even higher speed computational processors have made the threats of system "crackers," data theft, data corruption very real. This paper outlines some of the problems of current password security by demonstrating the ease by which individual accounts may be broken. Various techniques used by crackers are outlined, and finally one solution to this point of system vulnerability, a proactive password checker, is proposed.

1. Introduction

The security of accounts and passwords has always been a concern for the developers and users of Unix. When Unix was younger, the password encryption algorithm was a simulation of the M-209 cipher machine used by the U.S. Army during World War II [Morris1979]. This was a fair encryption mechanism in that it was difficult to invert under the proper circumstances, but suffered in that it was too fast an algorithm. On a PDP-11/70, each encryption took approximately 1.25ms, so that it was possible to check roughly 800 passwords/second. Armed with a dictionary of 250,000 words, a cracker could compare their encryptions with those all stored in the password file in a little more than five minutes. Clearly, this was a security hole worth filling.

In later (post-1976) versions of Unix, the DES algorithm [DES1975] was used to encrypt passwords. The user's password is used as the DES key, and the algorithm is used to encrypt a constant. The algorithm is iterated 25 times, with the result being an 11 character string plus a 2-character "salt." This method is similarly difficult to decrypt (further complicated through the introduction of one of 4096 possible salt values) and had the added advantage of being slow. On a μVAX-II (a machine substantially faster than a PDP-11/70), a single encryption takes on the order of 280ms, so that a determined cracker can only check approximately 3.6 encryptions a second. Checking this same dictionary of 250,000 words would now take over 19 *hours* of CPU time. Although this is still not very much time to break a single account, there is no guarantee that this account will use one of these words as a password. Checking the passwords on a system with 50 accounts would take on average 40 CPU *days* (since the random selection of salt values practically guarantees that each user's password will be encrypted with a different salt), with no guarantee of success. If this new, slow algorithm was combined with the user education needed to prevent the selection of obvious passwords, the problem seemed solved.

[†] This work was sponsored in part by the U.S. Department of Defense.

This paper was presented at the United Kingdom UNIX systems User Group (UKUUG) Summer 1990 Conference in London, UK.

Regrettably, two recent developments and the recurrence of an old one have brought the problem of password security back to the fore.

1) CPU speeds have gotten increasingly faster since 1976, so much so that processors that are 25-40 times faster than the PDP-11/70 (e.g., the DECstation 3100 used in this research) are readily available as desktop workstations. With inter-networking, many sites have hundreds of the individual workstations connected together, and enterprising crackers are discovering that the "divide and conquer" algorithm can be extended to multiple processors, especially at night when those processors are not otherwise being used. Literally thousands of times the computational power of 10 years ago can be used to break passwords.

2) New implementations of the DES encryption algorithm have been developed, so that the time it takes to encrypt a password and compare the encryption against the value stored in the password file has dropped below the 1ms mark [Bishop1988, Feldmeier1989]. On a single workstation, the dictionary of 250,000 words can once again be cracked in under five minutes. By dividing the work across multiple workstations, the time required to encrypt these words against all 4096 salt values could be no more than an hour or so.

3) Users are rarely, if ever, educated as to what are wise choices for passwords. If a password is in a dictionary, it is extremely vulnerable to being cracked, and users are simply not coached as to "safe" choices for passwords. Of those users who are so educated, many think that simply because their password is not in */usr/dict/words*, it is safe from detection. Many users also say that because they do not have any private files on-line, they are not concerned with the security of their account, little realizing that by providing an entry point to the system they allow damage to be wrought on their entire system by a malicious cracker.

Because the entirety of the password file is readable by all users, the encrypted passwords are vulnerable to cracking, both on-site and off-site. Many sites have responded to this threat with a reactive solution – they scan their own password files and advise those users whose passwords they are able to crack. The problem with this solution is that while the local site is testing its security, the password file is still vulnerable from the outside. The other problems, of course, are that the testing is very time consuming and only reports on those passwords it is able to crack. It does nothing to address user passwords which fall outside of the specific test cases (e.g., it is possible for a user to use as a password the letters "qwerty" – if this combination is not in the in-house test dictionary, it will not be detected, but there is nothing to stop an outside cracker from having a more sophisticated dictionary!).

Clearly, one solution to this is to either make */etc/passwd* unreadable, or to make the encrypted password portion of the file unreadable. Splitting the file into two pieces – a readable */etc/passwd* with all but the encrypted password present, and a "shadow password" file that is only readable by **root** is the solution proposed by Sun Microsystems (and others) that appears to be gaining popularity. It seems, however, that this solution will not reach the majority of non-Sun systems for quite a while, nor even, in fact, many Sun systems, due to many sites' reluctance to install new releases of software.[†]

What I propose, therefore, is a publicly available *proactive* password checker, which will enable users to change their passwords, and to check *a priori* whether the new password is "safe." The criteria for safety should be tunable on a per-site basis, depending on the degree of security desired. For example, it should be possible to specify a minimum length password, a restriction that only lower case letters are not allowed, that a password that looks like a license plate be illegal, and so on. Because this proactive checker will deal with the pre-encrypted passwords, it will be able to perform more sophisticated pattern matching on the password, and will be able to test the safety without having to go through the effort of cracking the encrypted version. Because the checking will be done automatically, the process of education can be transferred to the machine, which will instruct the user *why* a particular choice of password is bad.

[†] The problem of lack of password security is not just endemic to Unix. A recent Vax/VMS worm had great success by simply trying the username as the password. Even though the VMS user authorization file is inaccessible to ordinary users, the cracker simply tried a number of "obvious" password choices – and easily gained access.

2. Password Vulnerability

It has long been known that all a cracker need do to acquire access to a Unix machine is to follow two simple steps, namely:

1) Acquire a copy of that site's */etc/passwd* file, either through an unprotected *uucp* link, well known holes in *sendmail*, or via *ftp* or *tftp*.

2) Apply the standard (or a sped-up) version of the password encryption algorithm to a collection of words, typically */usr/dict/words* plus some permutations on account and user names, and compare the encrypted results to those found in the purloined */etc/passwd* file.

If a match is found (and often at least one will be found), the cracker has access to the targeted machine. Certainly, this mode of attack has been known for some time [Spafford1988], and the defenses against this attack have also long been known. What is lacking from the literature is an accounting of just how vulnerable sites are to this mode of attack. In short, many people know that there is a problem, but few people believe it applies to them.

"There is a fine line between helping administrators protect their systems and providing a cookbook for bad guys." [Grampp1984] The problem here, therefore, is how to divulge useful information on the vulnerability of systems, without providing too much information, since almost certainly this information could be used by a cracker to break into some as-yet unviolated system. Most of the work that I did was of a general nature – I did not focus on a particular user or a particular system, and I did not use any personal information that might be at the disposal of a dedicated "bad guy." Thus any results which I have been able to garner indicate only general trends in password usage, and cannot be used to great advantage when breaking into a particular system. This generality notwithstanding, I am sure that any self-respecting cracker would already have these techniques at their disposal, and so I am not bringing to light any great secret. Rather, I hope to provide a basis for protection for systems that can guard against future attempts at system invasion.

2.1. The Survey and Initial Results

In October and again in December of 1989, I asked a number of friends and acquaintances around the United States and Great Britain to participate in a survey. Essentially what I asked them to do was to mail me a copy of their */etc/passwd* file, and I would try to crack their passwords (and as a side benefit, I would send them a report of the vulnerability of their system, although at no time would I reveal individual passwords nor even of their sites participation in this study). Not surprisingly, due to the sensitive nature of this type of disclosure, I only received a small fraction of the replies I hoped to get, but was nonetheless able to acquire a database of nearly 15,000 account entries. This, I hoped, would provide a representative cross section of the passwords used by users in the community.

Each of the account entries was tested by a number of intrusion strategies, which will be covered in greater detail in the following section. The possible passwords that were tried were based on the user's name or account number, taken from numerous dictionaries (including some containing foreign words, phrases, patterns of keys on the keyboard, and enumerations), and from permutations and combinations of words in those dictionaries. All in all, after nearly 12 CPU months of rather exhaustive testing, approximately 25% of the passwords had been guessed. So that you do not develop a false sense of security too early, I add that 21% (nearly 3,000 passwords) were guessed in the first week, and that in the first 15 minutes of testing, 368 passwords (or 2.7%) had been cracked using what experience has shown would be the most fruitful line of attack (i.e., using the user or account names as passwords). These statistics are frightening, and well they should be. On an average system with 50 accounts in the */etc/passwd* file, one could expect the first account to be cracked in under 2 minutes, with 5–15 accounts being cracked by the end of the first day. Even though the **root** account may not be cracked, all it takes is one account being compromised for a cracker to establish a toehold in a system. Once that is done, any of a number of other well-known security loopholes (many of which have been published on the network) can be used to access or destroy any information on the machine.

It should be noted that the results of this testing do not give us any indication as to what the *uncracked* passwords are. Rather, it only tells us what was essentially already known – that users are likely to use words that are familiar to them as their passwords [Riddle1989]. What new information it did provide, however, was the *degree* of vulnerability of the systems in question, as well as providing a basis for developing a proactive password changer – a system which pre-checks a password before it is entered into the system, to determine whether that password will be vulnerable to this type of attack. Passwords which can be derived from a dictionary are clearly a bad idea [Alvare1988], and users should be prevented from using them. Of course, as part of this censoring process, users should also be told *why* their proposed password is not good, and what a good class of password would be.

As to those passwords which remain unbroken, I can only conclude that these are much more secure and "safe" than those to be found in my dictionaries. One such class of passwords is word pairs, where a password consists of two short words, separated by a punctuation character. Even if only words of 3 to 5 lower case characters are considered, */usr/dict/words* provides 3000 words for pairing. When a single intermediary punctuation character is introduced, the sample size of 90,000,000 possible passwords is rather daunting. On a DECstation 3100, testing each of these passwords against that of a single user would require over 25 CPU *hours* – and even then, no guarantee exists that this is the type of password the user chose. Introducing one or two upper case characters into the password raises the search set size to such magnitude as to make cracking untenable.

Another "safe" password is one constructed from the initial letters of an easily remembered, but not too common phrase. For example, the phrase "Unix is a trademark of Bell Laboratories" could give rise to the password "UiatoBL." This essentially creates a password which is a random string of upper and lower case letters. Exhaustively searching this list at 1000 tests per second with only 6 character passwords would take nearly 230 CPU days. Increasing the phrase size to 7 character passwords makes the testing time over 32 CPU *years* – a Herculean task that even the most dedicated cracker with huge computational resources would shy away from.

Thus, although I don't know what passwords were chosen by those users I was unable to crack, I can say with some surety that it is doubtful that anyone else could crack them in a reasonable amount of time, either.

2.2. Method of Attack

A number of techniques were used on the accounts in order to determine if the passwords used for them were able to be compromised. To speed up testing, all passwords with the same salt value were grouped together. This way, one encryption per password per salt value could be performed, with multiple string comparisons to test for matches. Rather than considering 15,000 accounts, the problem was reduced to 4,000 salt values. The password tests were as follows:

1) Try using the user's name, initials, account name, and other relevant personal information as a possible password. All in all, up to 130 different passwords were tried based on this information. For an account name **klone** with a user named "Daniel V. Klein," some of the passwords that would be tried were: klone, klone0, klone1, klone123, dvk, dvkdvk, dklein, DKlein, leinad, nielk, dvklein, danielk, DvkkvD, DANIEL-KLEIN, (klone), KleinD, etc.

2) Try using words from various dictionaries. These included lists of men's and women's names (some 16,000 in all); places (including permutations so that "spain," "spanish," and "spaniard" would all be considered); names of famous people; cartoons and cartoon characters; titles, characters, and locations from films and science fiction stories; mythical creatures (garnered from Bulfinch's mythology and dictionaries of mythical beasts); sports (including team names, nicknames, and specialized terms); numbers (both as numerals – "2001," and written out – "twelve"); strings of letters and numbers ("a," "aa," "aaa," "aaaa," etc.); Chinese syllables (from the Pinyin Romanization of Chinese, a international standard system of writing Chinese on an English keyboard); the King James Bible; biological terms; common and vulgar phrases (such as "fuckyou," "ibmsux," and "deadhead"); keyboard patterns (such as "qwerty," "asdf," and "zxcvbn"); abbreviations (such as "roygbiv" – the colors in the rainbow, and "ooottafagvah" – a mnemonic for remembering the 12 cranial nerves); machine names (acquired from */etc/hosts*); characters, plays, and locations from Shakespeare; common Yiddish words; the names of asteroids; and a collection of words from various technical papers I had previously published. All told, more than 60,000

separate words were considered per user (with any inter- and intra-dictionary duplicates being discarded).

3) Try various permutations on the words from step 2. This included making the first letter upper case or a control character, making the entire word upper case, reversing the word (with and without the aforementioned capitalization), changing the letter 'o' to the digit '0' (so that the word "scholar" would also be checked as "sch0lar"), changing the letter 'l' to the digit '1' (so that "scholar" would also be checked as "scho1ar," and also as "sch01ar"), and performing similar manipulations to change the letter 'z' into the digit '2', and the letter 's' into the digit '5'. Another test was to make the word into a plural (irrespective of whether the word was actually a noun), with enough intelligence built in so that "dress" became "dresses," "house" became "houses," and "daisy" became "daisies." We did not consider pluralization rules exhaustively, though, so that "datum" forgivably became "datums" (not "data"), while "sphynx" became "sphynxs" (and not "sphynges"). Similarly, the suffixes "-ed," "-er," and "-ing" were added to transform words like "phase" into "phased," "phaser," and "phasing." These 14 to 17 additional tests per word added another 1,000,000 words to the list of possible passwords that were tested for each user.

4) Try various capitalization permutations on the words from step 2 that were not considered in step 3. This included all single letter capitalization permutations (so that "michael" would also be checked as "mIchael," "miChael," "micHael," "michAel," etc.), double letter capitalization permutations ("MIchael," "MiChael," "MicHael," ... , "mIChael," "mIcHael," etc.), triple letter permutations, and so on. The single letter permutations added roughly another 400,000 words to be checked per user, while the double letter permutations added another 1,500,000 words. Three letter permutations would have added at least another 3,000,000 words *per user* had there been enough time to complete the tests. Tests of 4, 5, and 6 letter permutations were deemed to be impracticable without much more computational horsepower to carry them out.

5) Try foreign language words on foreign users. The specific test that was performed was to try Chinese language passwords on users with Chinese names. The Pinyin Romanization of Chinese syllables was used, combining syllables together into one, two, and three syllable words. Because no tests were done to determine whether the words actually made sense, an exhaustive search was initiated. Since there are 398 Chinese syllables in the Pinyin system, there are 158,404 two syllable words, and slightly more than 16,000,000 three syllable words.† A similar mode of attack could as easily be used with English, using rules for building pronounceable nonsense words.

6) Try word pairs. The magnitude of an exhaustive test of this nature is staggering. To simplify this test, only words of 3 or 4 characters in length from */usr/dict/words* were used. Even so, the number of word pairs is $O(10^7)$ (multiplied by 4096 possible salt values), and as of this writing, the test is only 10% complete.

For this study, I had access to four DECstation 3100's, each of which was capable of checking approximately 750 passwords per second. Even with this total peak processing horsepower of 3,000 tests per second (some machines were only intermittently available), testing the $O(10^{10})$ password/salt pairs for the first four tests required on the order of 12 CPU *months* of computations. The remaining two tests are still ongoing after an additional 18 CPU months of computation. Although for research purposes this is well within acceptable ranges, it is a bit out of line for any but the most dedicated and resource-rich cracker.

† The astute reader will notice that 398^3 is in fact 63,044,972. Since Unix passwords are truncated after 8 characters, however, the number of unique polysyllabic Chinese passwords is only around 16,000,000. Even this reduced set was too large to complete under the imposed time constraints.

2.3. Summary of Results

The problem with using passwords that are derived directly from obvious words is that when a user thinks "Hah, no one will guess this permutation," they are almost invariably wrong. Who would ever suspect that I would find their passwords when they chose "fylgjas" (guardian creatures from Norse mythology), or the Chinese word for "hen-pecked husband"? No matter what words or permutations thereon are chosen for a password, if they exist in some dictionary, they are susceptible to directed cracking. The following table give an overview of the types of passwords which were found through this research.

A note on the table is in order. The number of matches given from a particular dictionary is the total number of matches, irrespective of the permutations that a user may have applied to it. Thus, if the word "wombat" were a particularly popular password from the biology dictionary, the following table will not indicate whether it was entered as "wombat," "Wombat," "TABMOW," "w0mbat," or any of the other 71 possible differences that this research checked. In this way, detailed information can be divulged without providing much knowledge to potential "bad guys."

Additionally, in order to reduce the total search time that was needed for this research, the checking program eliminated both inter- and intra-dictionary duplicate words. The dictionaries are listed in the order tested, and the total size of the dictionary is given in addition to the number of words that were eliminated due to duplication. For example, the word "georgia" is both a female name and a place, and is only considered once. A password which is identified as being found in the common names dictionary might very well appear in other dictionaries. Additionally, although "duplicate," "duplicated," "duplicating" and "duplicative" are all distinct words, only the first eight characters of a password are used in Unix, so all but the first word are discarded as redundant.

Passwords cracked from a sample set of 13,797 accounts

Type of Password	Size of Dictionary	Duplicates Eliminated	Search Size	# of Matches	Pct. of Total	Cost/Benefit Ratio*
User/account name	130†	–	130	368	2.7%	2.830
Character sequences	866	0	866	22	0.2%	0.025
Numbers	450	23	427	9	0.1%	0.021
Chinese	398	6	392	56	0.4%‡	0.143
Place names	665	37	628	82	0.6%	0.131
Common names	2268	29	2239	548	4.0%	0.245
Female names	4955	675	4280	161	1.2%	0.038
Male names	3901	1035	2866	140	1.0%	0.049
Uncommon names	5559	604	4955	130	0.9%	0.026
Myths & legends	1357	111	1246	66	0.5%	0.053
Shakespearean	650	177	473	11	0.1%	0.023
Sports terms	247	9	238	32	0.2%	0.134
Science fiction	772	81	691	59	0.4%	0.085
Movies and actors	118	19	99	12	0.1%	0.121
Cartoons	133	41	92	9	0.1%	0.098
Famous people	509	219	290	55	0.4%	0.190
Phrases and patterns	998	65	933	253	1.8%	0.271
Surnames	160	127	33	9	0.1%	0.273
Biology	59	1	58	1	0.0%	0.017
/usr/dict/words	24474	4791	19683	1027	7.4%	0.052
Machine names	12983	3965	9018	132	1.0%	0.015
Mnemonics	14	0	14	2	0.0%	0.143
King James bible	13062	5537	7525	83	0.6%	0.011
Miscellaneous words	8146	4934	3212	54	0.4%	0.017
Yiddish words	69	13	56	0	0.0%	0.000
Asteroids	3459	1052	2407	19	0.1%	0.007
Total	86280	23553	62727	**3340**	**24.2%**	0.053

Length of Cracked Passwords		
Length	**Count**	**Percentage**
1 character	4	0.1%
2 characters	5	0.2%
3 characters	66	2.0%
4 characters	188	5.7%
5 characters	317	9.5%
6 characters	1160	34.7%
7 characters	813	24.4%
8 characters	780	23.4%

The results of the word-pair tests are not included in either of the two tables. However, at the time of this writing, the test was approximately 10% completed, having found an additional 0.4% of the passwords in the sample set. It is probably reasonable to guess that a total of 4% of the passwords would be cracked by using word pairs.

3. Action, Reaction, and Proaction

What then, are we to do with the results presented in this paper? Clearly, something needs to be done to safeguard the security of our systems from attack. It was with intention of enhancing security that this study was undertaken. By knowing what kind of passwords users use, we are able to prevent them from using those that are easily guessable (and thus thwart the cracker).

One approach to eliminating easy-to-guess passwords is to periodically run a password checker – a program which scans */etc/passwd* and tries to break the passwords in it [Raleigh1988]. This approach has two major drawbacks. The first is that the checking is very time consuming. Even a system with only 100 accounts can take over a month to diligently check. A halfhearted check is almost as bad as no check at all, since users will find it easy to circumvent the easy checks and still have vulnerable passwords. The second drawback is that it is very resource consuming. The machine which is being used for password checking is not likely to be very useful for much else, since a fast password checker is also extremely CPU intensive.

Another popular approach to eradicating easy-to-guess passwords is to force users to change their passwords with some frequency. In theory, while this does not actually eliminate any easy-to-guess passwords, it prevents the cracker from dissecting */etc/passwd* "at leisure," since once an account is broken, it is likely that that account will have had it's password changed. This is of course, only theory. The biggest disadvantage is that there is usually nothing to prevent a user from changing their password from "Daniel" to "Victor" to "Klein" and back again (to use myself as an example) each time the system demands a new password. Experience has shown that even when this type of password cycling is precluded, users are easily able to circumvent simple tests by using easily remembered (and easily guessed) passwords such as "dvkJanuary," "dvkFebruary," etc [Reid1989]. A good password is one that is easily remembered, yet difficult to guess. When confronted with a choice between remembering a password or creating one that is hard to guess, users will almost always opt for the easy way out, and throw security to the wind.

Which brings us to the third popular option, namely that of assigned passwords. These are often words from a dictionary, pronounceable nonsense words, or random strings of characters. The problems here are numerous and manifest. Words from a dictionary are easily guessed, as we have seen.

* In all cases, the cost/benefit ratio is the number of matches divided by the search size. The more words that needed to be tested for a match, the lower the cost/benefit ratio.

† The dictionary used for user/account name checks naturally changed for each user. Up to 130 different permutations were tried for each.

‡ While monosyllablic Chinese passwords were tried for all users (with 12 matches), polysyllabic Chinese passwords were tried only for users with Chinese names. The percentage of matches for this subset of users is 8% – a greater hit ratio than any other method. Because the dictionary size is over 16×10^6, though, the cost/benefit ratio is infinitesimal.

Pronounceable nonsense words (such as "trobacar" or "myclepate") are often difficult to remember, and random strings of characters (such as "h3rT+aQz") are even harder to commit to memory. Because these passwords have no personal mnemonic association to the users, they will often write them down to aid in their recollection. This immediately discards any security that might exist, because now the password is visibly associated with the system in question. It is akin to leaving the key under the door mat, or writing the combination to a safe behind the picture that hides it.

A fourth method is the use of "smart cards." These credit card sized devices contain some form of encryption firmware which will "respond" to an electronic "challenge" issued by the system onto which the user is attempting to gain acccess. Without the smart card, the user (or cracker) is unable to respond to the challenge, and is denied access to the system. The problems with smart cards have nothing to do with security, for in fact they are very good warders for your system. The drawbacks are that they can be expensive and must be carried at all times that access to the system is desired. They are also a bit of overkill for research or educational systems, or systems with a high degree of user turnover.

Clearly, then, since all of these systems have drawbacks in some environments, an additional way must be found to aid in password security.

3.1. A Proactive Password Checker

The best solution to the problem of having easily guessed passwords on a system is to prevent them from getting on the system in the first place. If a program such as a password checker *reacts* by detecting guessable passwords already in place, then although the security hole is found, the hole existed for as long as it took the program to detect it (and for the user to again change the password). If, however, the program which changes user's passwords (i.e., */bin/passwd*) checks for the safety and guessability *before* that password is associated with the user's account, then the security hole is never put in place.

In an ideal world, the proactive password changer would require eight character passwords which are not in any dictionary, with at least one control character or punctuation character, and mixed upper and lower case letters. Such a degree of security (and of accompanying inconvenience to the users) might be too much for some sites, though. Therefore, the proactive checker should be tuneable on a per-site basis. This tuning could be accomplished either through recompilation of the *passwd* program, or more preferably, through a site configuration file.

As distributed, the behavior of the proactive checker should be that of attaining maximum password security – with the system administrator being able to turn off certain checks. It would be desireable to be able to test for and reject all password permutations that were detected in this research (and others), including:

- Passwords based on the user's account name
- Passwords which exactly match a word in a dictionary (not just */usr/dict/words*)
- Passwords which match a reversed word in the dictionary
- Passwords which match a word in a dictionary with an arbitrary letter turned into a control character
- Passwords which are simple conjugations of a dictionary word (i.e., plurals, adding "ing" or "ed" to the end of the word, etc.)
- Passwords based on the user's initials or given name
- Passwords which match a word in the dictionary with some or all letters capitalized
- Passwords which match a reversed word in the dictionary with some or all letters capitalized
- Passwords which match a dictionary word with the numbers '0', '1', '2', and '5' substituted for the letters 'o', 'l',
- Passwords which are patterns from the keyboard (i.e., "aaaaaa" or "qwerty")

- Passwords which are shorter than a specific length (i.e., nothing shorter than six characters)
- Passwords which do not contain mixed upper and lower case, or mixed letters and numbers, or mixed letters and punctuation
- Passwords which consist solely of numeric characters (i.e., Social Security numbers, telephone numbers, house addresses or office numbers)
- Passwords which look like a state-issued license plate number

The configuration file which specifies the level of checking need not be readable by users. In fact, making this file unreadable by users (and by potential crackers) enhances system security by hiding a valuable guide to what passwords *are* acceptable (and conversely, which kind of passwords simply cannot be found).

Of course, to make this proactive checker more effective, it woule be necessary to provide the dictionaries that were used in this research (perhaps augmented on a per-site basis). Even more importantly, in addition to rejecting passwords which could be easily guessed, the proactive password changer would also have to tell the user *why* a particular password was unacceptable, and give the user suggestions as to what an acceptable password looks like.

4. Conclusion (and Sermon)

It has often been said that "good fences make good neighbors." On a Unix system, many users also say that "I don't care who reads my files, so I don't need a good password." Regrettably, leaving an account vulnerable to attack is not the same thing as leaving files unprotected. In the latter case, all that is at risk is the data contained in the unprotected files, while in the former, the whole system is at risk. Leaving the front door to your house open, or even putting a flimsy lock on it, is an invitation to the unfortunately ubiquitous people with poor morals. The same holds true for an account that is vulnerable to attack by password cracking techniques.

While it may not be actually true that good fences make good neighbors, a good fence at least helps keep out the bad neighbors. Good passwords are equivalent to those good fences, and a proactive checker is one way to ensure that those fences are in place *before* a breakin problem occurs.

References

Morris1979.
Robert T. Morris and Ken Thompson, "Password Security: A Case History," *Communications of the ACM*, vol. 22, no. 11, pp. 594-597, November 1979.

DES1975.
"Proposed Federal Information Processing Data Encryption Standard," *Federal Register (40FR12134)*, March 17, 1975.

Bishop1988.
Matt Bishop, "An Application of a Fast Data Encryption Standard Implementation," *Computing Systems*, vol. 1, no. 3, pp. 221-254, Summer 1988.

Feldmeier1989.
David C. Feldmeier and Philip R. Karn, "UNIX Password Security – Ten Years Later," *CRYPTO Proceedings*, Summer 1989.

Spafford1988.
Eugene H. Spafford, "The Internet Worm Program: An Analysis," Purdue Technical Report CSD-TR-823, Purdue University, November 29, 1988.

Grampp1984.
F. Grampp and R. Morris, "Unix Operating System Security," *AT&T Bell Labs Technical Journal*, vol. 63, no. 8, pp. 1649-1672, October 1984.

Riddle1989.

Bruce L. Riddle, Murray S. Miron, and Judith A. Semo, ''Passwords in Use in a University Timesharing Environment,'' *Computers & Security*, vol. 8, no. 7, pp. 569-579, November 1989.

Alvare1988.

Ana Marie De Alvare and E. Eugene Schultz, Jr., ''A Framework for Password Selection,'' *USENIX UNIX Security Workshop Proceedings*, August 1988.

Raleigh1988.

T. Raleigh and R. Underwood, ''CRACK: A Distributed Password Advisor,'' *USENIX UNIX Security Workshop Proceedings*, August 1988.

Reid1989.

Dr. Brian K Reid, DEC Western Research Laboratory, 1989. Personal communication.

An Extendable Password Checker

Matt Bishop

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

EXTENDED ABSTRACT

Computer generation of passwords is a delicate art: the passwords cannot be random, for if they are users will write them down, yet they cannot be too non-random, for otherwise they could be easily guessed. The use of pronounceable passwords tries to strike the balance, yet even these are often difficult to remember. Hence user selection of passwords is in widespread use; unfortunately, many users select easy to guess passwords. In a series of experiments in 1979, roughly 33% of the passwords appeared in a dictionary and an astonishing 86% of all passwords were guessed; in similar experiments conducted last year at the Software Engineering Institute, over 21% of the passwords from many different sites were guessed correctly within one week, and roughly 3% were found in 15 minutes. Clearly, this is as unacceptable as users writing down their passwords.

We introduce a tool, called a *proactive password checker*, designed to ameliorate this problem at the source. When a user uses this tool to change (or set) a password, the proactive password checker runs a number of tests on the proposed password to see if it can be guessed easily; if so, the user is informed and the password is rejected. Thus, the tool allows users to select passwords they can remember, but prevents them from choosing passwords which are deemed unsafe.

We describe the design and implementation of a UNIX™-based version of this tool. The novel feature of this proactive password changer is the use of a very flexible, very powerful interpreted little language for representing tests for guessability. (That the language is interpreted means the tests can be changed without recompilation.) The little language describes tests with associated error messages; each test is composed of any combination of variables, constants (numbers and strings), equality and pattern matching relations, and file and program output scanning instructions.

The variables are either initialized within the configuration file or set from the user's proposed password, and contain useful information about the characteristics of the proposed password and the environment (host, domain, user). The reference mechanism allows various lexicographic transformations of the contents of any string-valued variable to be used.

The values of variables may be compared to one another, or to constant numbers or strings, for strict equality or for pattern matches. Because of the complexity of most pattern matching expressions, the pattern matchers use the system's standard library routines to determine if a value matches a pattern; this means that a system administrator who uses a text editor need not learn a new pattern matching system.

File and program scanning instructions compare a value to the contents of a file or the output of any program, or match the contents of the file or output of the program to a pattern written in the style of the system's text editor. Comparison is done line by line, and if any line satisfies the

UNIX™ is a registered trademark of AT&T Bell Laboratories relationship, the test succeeds. Variables in the file names and program commands are replaced before the files or commands are executed, so it is easy to condition the test, the files, and the programs upon the user's or group's identity.

Error messages may follow each test; if any test succeeds, the appropriate error message is printed. This allows users to be told precisely why their selection is unacceptable, rather than limiting the response to a more generic "`password not suitable; try again.`"

Some sample tests, associated error messages, and the proposed passwords making the tests succeed (and hence the passwords will be rejected), are:

```
"%p"=~"[0-9][A-Za-z]{3}[0-9]{3}""can't use license plate numbers"
```

succeeds if the password looks like a California automobile license plate number (a digit followed by three letters followed by three digits); it matches the proposed password (`%p`) against an appropriate pattern;

```
"%*p"=~"\(%u\)*"           "can't use (repeated) login name"
```

succeeds if the password is 0 or more repetitions of the user's login name; it first maps all alphabetic characters in the proposed password to lower case (`%*p`), then matches them against the pattern composed of the login name (`%u`) repeated 0 or more times;

```
"%^p"=="%-^l"               "cannot use last name reversed"
```

succeeds if the password is the same as the user's last name reversed; this test makes all alphabetic characters in the proposed password upper case (`%^p`), does the same for the reversal of the user's last name (`%-^l`), and tests for equality.

```
"%p"=~"^[a-z]*$"|"%p"=~"^[A-Z]*$""if all letters, must mix case"
(%#p==%b)&(%v==0)                "if all letters, must mix case"
```

either test succeeds if the proposed password is a monocase alphabetic sequence; the first just matches the proposed password against two appropriate patterns, and the second compares the length of the proposed password (`%#p`) to the number of alphabetic characters in it (`%b`), and if the password is composed *only* of alphabetic characters, tests if they are all of the same case (`%v` is 1 if so);

```
{echo %p | spell}==""            "cannot use word in dictionary"
```

succeeds if the word is an English word; the password is given as input to the system spelling checker; if found, the checker returns nothing. This catches words like "`waters`" which are often not in system dictionaries (as it is clearly a plural of "`water`").

We shall discuss other features of the proactive password changer such as the assignment of user information to specific variables, coping with passwords of limited length, the logging system which is used to monitor the usefulness of the tests for guessability, and present some experiences of a large site (50-100 computers of various manufactures and running a variety of UNIX™-based operating systems) at which this tool has been in production use, especially on the installation and maintenance of the configuration file. We shall close with some problems and suggestions for future work.

Password Security
In a Large Distributed Environment

Michele D. Crabb
Computer Sciences Corporation*
NASA Ames Research Center
Moffett Field, CA 94035

ABSTRACT

With the increased interest in, or rather need for, better security in UNIX† computing environments, password security and control is one issue which must be addressed completely. In the past, when a single VAX 11/780‡ serving 30 users was the status quo, password security was simple in that a system administrator merely had to remember a few privileged passwords and ensure that all user accounts had passwords. However, in today's large-scale environments where central support of 100 or more systems is common and with the availability of fast password cracking programs, password security has become a complex issue. Password security no longer involves just remembering the root password for a particular system or ensuring all accounts have passwords. It now involves the concept of using "smart" passwords, management of a large number of privileged passwords and a equally large number of people who need access to those passwords, and a much tighter control on who has access to the *root* and other privileged accounts.

This paper will examine the current methods used for password security and control at the Numerical Aerodynamic Simulation (NAS) facility at NASA-Ames Research Center. The NAS environment consists of over 170 systems running the UNIX operating system with the TCP/IP network protocol software which supports over 1000 users nationwide. On-going support and software development for NAS is provided by close to 150 personnel. Due to the large number of systems, users and support personnel, implementing password security and control at NAS has been a challenging task.

Some specific topics that will be discussed are: the concept of "special access" accounts; how password groups and levels of access are defined; the formal NAS policies concerning access to privileged accounts; how all the information regarding the numerous privileged passwords is tracked and recorded; and one alternative to providing the *root* password to everyone who needs root access.

* This work is supported by contract NAS2-12961 of the National Aeronautics and Space Administration to Computer Sciences Corporation.

† UNIX is a registered trademark of AT&T.

‡ VAX is a trademark of Digital Equipment Corporation.

1.0 Introduction

The Numerical Aerodynamic Simulation (NAS) facility, which supports over 1000 users nationwide, is composed of over 170 computer systems running some version of the UNIX operating system with TCP/IP network protocol software. These systems are interconnected into a heterogeneous network using various types of networking hardware. Ongoing support and software development for the NAS facility is provided by close to 150 personnel. Due to the nature of their responsibilities, many of these people require access to the privileged accounts on one or more of the NAS systems. With numbers such as these, it becomes quite apparent that password security [1] and control in a large environment like NAS can be a challenging task.

This paper will discuss the methods currently used for providing password security and control for special accounts such as the *root* account. The notion of a "special access account" is defined along with policies governing such accounts. A step-by-step description of a "day in the life of a password change at NAS" is provided and one alternative to sharing of the *root* password is examined. Some ideas for the future improvements to password security are also presented.

2.0 A Brief Look into the Past

Up until three or four years ago, password security at NAS was lax at best. The environment was more of the type found in a small university than in a national supercomputing center. No one person had knowledge of all the people who had access to the *root* account on the various systems. If a person needed the *root* password for a certain system, he/she simply asked around until a person was found who knew the *root* password. During this time, it was also common to have *root* passwords that were not known by anyone. Use of *root* ***/.rhosts*** files was widespread; therefore, if the *root* password was not known for a particular system, you merely had to remotely log in from another system where the *root* password was known. Periodic password changes were also a rare event. When the *root* passwords were changed, people were not notified in any formal manner. If a person discovered the password had changed, he/she would ask a system analyst or control room analyst for the new password.

As the number of NAS systems and users increased, password security started to become an issue. Primitive methods for tracking *root* passwords across all NAS systems were put into place. One early solution to the problem was to use the **crypt (1)** command to encrypt an ASCII file which contained all of the *root* passwords. Then, instead of giving out the individual passwords, the key used to encrypt the file was given out. However, this method was vulnerable to attacks due to the weaknesses of the **crypt (1)** command and the availability of the "Cryptographer's Workbench" package [2]. As time passed, password security slowly increased to its current level, which will be described in the remainder of this paper.

3.0 Password Security for Special Access Accounts

Before discussing password security for special access accounts, I need to first define what is meant by a special access account and provide a brief description of the NAS system support and administration philosophy. A "special access" account is defined to be any account required for the support of the NAS facility and whose password is centrally controlled. A special access account may be shared by numerous support personnel or not shared at all. Any person who has the password and privilege to use one of the special access accounts is considered to have "special access."

3.1 The NAS Support Philosophy

The philosophy for system support at NAS is probably much like many large computing environments. To facilitate support of NAS systems, the different branches of the NAS Systems Division are subdivided into "subsystems" based on computer architecture or type of support (i.e., on-going vs. development). Some of the current subsystems are: Workstation subsystem (WKS), High Speed Processor subsystem (HSP) and Data Communication Network Support subsystem (DCN).

Support responsibilities are aligned via the different subsystems in the NAS Systems Division. Generally, there will be one Lead System Analyst (LSA) and one or more support personnel providing system administration and software development in any one subsystem. A LSA or support analyst for one subsystem does not have any authority to make changes on a system in another subsystem, even if that person has *root* privilege on the other system. A minimum requirement for each support person is to have access to any privileged account on the systems within their area of responsibility. Some people provide support in multiple areas and thus require a wider range of special access. For example, Control Room Analysts are responsible for front line support of all NAS systems and must have access to all special access accounts.

3.2 Password Groups and Levels of Access

At NAS just keeping tabs on the *root* passwords for all systems and who has access to these passwords is a major task. Throw in an additional dozen or so special access accounts which may be on one or all NAS systems and you have one major password security headache. To simplify the task of maintaining *root* passwords on 180 plus systems, NAS has had to do away with some of the traditional rules governing *root* and other special account passwords. At NAS we use the concept of password groups where systems with similar support status and/or of similar architecture are named as a password group and share a common *root* password. For example, all fully supported production file servers would share a common *root* password referred to as the "Sun FS" password. All fully supported production workstations (there are three architectures in this group) would share a common *root* password called "WKS Pro."

Password groups are only used for groups of systems which are supported by people who can have access on all systems in the password group. Therefore, if a group of systems is supported by two or more people who do not have access to all privileged accounts on those systems, each system would be required to have a separate *root* password. Furthermore, any system which cannot be classified into one of the common support classifications will have a unique *root* password. Shared support accounts, such as a printer administration account, would have a common password across all NAS systems.

Closely related to password groups for systems are the levels of access for support groups. Most support groups at NAS are aligned with a NAS subsystem (i.e., WKS support group, VAX support group, HSP support group). In order to reduce the amount of special access required by system support personnel, a minimum baseline of special access groups for individual support groups has been established. In other words, for any one support group, there is a baseline group of special access passwords required by all members of the support group. For example, all members of the HSP support group need access to the *root* account on all HSP production systems and the HSP gateway systems. There are some support groups which require access to the *root* account on every system at NAS, due to the scope of their responsibilities. The Network Support Group and the Control Room Analysts are two such groups.

3.3 Policies Concerning Special Access Accounts

Due to the increasing number of people with special access at the NAS facility, it was necessary to establish a policy concerning access to special (or privileged) accounts. The policy provides a set of requirements for the regulation and use of special access on the NAS systems. The policy also provides procedures for the addition and removal of people from the special access database and a mechanism for periodic reviews of the special access database.

Part of implementing the policy on special access was developing a separate form to request special access and writing a set of rules concerning the do's and don'ts of special access. The Special Access Request form requires users to justify their need for such access and it requires management signatures at three levels. On the form is a small set of basic rules governing special access. Persons requesting special access must sign the form certifying that they will abide by all rules and regulations concerning special access.

The special access form documents the "who, what, when, where and why" of special access for each user and serves as a helpful document for future reference. The "who" is the person requesting special access. The "what" is the type of special access they are requesting. The "when" is the date they were granted special access and the date that their special access expires or must be re-justified. The "where" is the system(s) on which they have requested special access. The "why", the most important of the five, is the user's justification for needing special access. Prior to the implementation of the special access policy, little justification was required when requesting access to privileged accounts. As a result, many people had access to accounts for which they had no justification or need.

The full set of rules concerning special access is documented separately and is referred to as the "Special Access Guidelines Agreement." The agreement lists some general guidelines as well as some specific do's and don'ts governing the use of special access. The main motivation for the Special Access Guidelines Agreement was to help people use their special access in a responsible and secure manner. The agreement also provides a handy reference point for those abusers of special access who use the excuse "they did not know better."

Some of the "do's" listed in the agreement are:

1) Be aware of the NAS environment.
2) If possible, log on to a system as yourself and then "su" to the needed UID.
3) Use special access only if necessary.
4) Document all major actions.
5) Have a backup plan in case something goes wrong.

Some of the specific "don'ts" listed in the agreement are:

1) Do not share special access passwords with anyone.
2) Do not write down the passwords or current algorithm.
3) Do not read or send personal E-mail, play games, read the net news or edit personal files using a special access account.
4) Do not browse or alter other users' files.

3.4 Tracking the Information

By now it should be apparent that there is a lot of information to store and to track. Just trying to remember all of the password group names can be as difficult as trying to remember the actual passwords, or the names of all the people with special access. Obviously, this is an area where a database is needed. At NAS, all of the password information is stored in a HyperCard database on a MacIntosh* diskette, which is kept under lock and key. Information stored in the database includes: names of special access groups, all current passwords and passwords for the last three months, and the full names and login IDs of all support personnel with special access.

For each support person there is associated a "card." The information contained on each card is the user's full name, the user's login ID, the contact person (for off-site employees), the current date, and list of password groups for which the user has been granted access. In addition to user cards, there are also password cards. Each password card contains a table of all password groups with the appropriate password for each group. A new password card is created each time the passwords are changed, which provides an audit trail of previous passwords used. Each time a new user or new password group is added to the database, a new list of users with special access is created. This list, called the "sulist," is a quick reference of who has access for what group of systems.

3.5 The Use of Password Sheets

Aside from the large amount of information which must be stored, a major motivation for implementing a password database was to facilitate the printing of "password sheets." Due to the large number of password groups, it is necessary to provide support personnel with a small sheet of paper which lists all of the passwords for which they have been granted access. Even if every person was able to remember up to 40 different passwords, there would still be the dilemma of informing 85 or more people each time the passwords are changed. The natural solution seemed to be to provide each person with their personal list of passwords which they could carry on their person.

The password sheet currently provided is small enough to fit in an ID badge envelope

* HyperCard and MacIntosh are trademarks of Apple Computer Corporation.

which can be worn along with the other NAS badges. Since the implementation of password sheets, the task of informing everyone of a password change has been reduced to having users come pick up their new password sheets when they need them. Even though the use of password sheets seems to nullify one of the major concepts of password security, it has served NAS well over the past three years.

To offset some of the dangers of having passwords documented on a sheet of paper, NAS has implemented the concept of using a password algorithm. A password algorithm is some function to be applied to the password written on the password sheet to get the actual password. So, in reality, the passwords are not written down verbatim. For example, on the password sheet the password listed for system "A" might be "freem3" and the algorithm might be "capitalize the first vowel and add a dash "-" to the end." Then the actual password for system "A" would be "frEem3-." The password algorithm is not to be written down, thus allowing NAS to meet two of the normal security validation checks. At NAS we can verify "what you have," which is the password as written on the password sheet. We can verify "what you know," which is the password algorithm needed to obtain actual passwords. But unfortunately, we cannot verify "what you are" with the current technology[3].

3.6 A Day In The Life of A Password Change

Special access passwords are changed on a periodic basis. The passwords are split into two major change groups - quarterly and monthly. Passwords for systems and/or accounts which are shared by a small group of people (< 10) are changed on a quarterly basis. All other passwords are changed on a monthly basis. The password algorithm is changed on a quarterly basis.

Given the large number of special access password groups, the very large number of systems at NAS and the number of people who receive special access passwords, the periodic change of passwords is not a trivial exercise. The entire exercise takes several days to complete and must be treated with care so as not to disclose the new passwords or to interrupt system service because a password was changed without notifying the appropriate people. The task can be broken down into several steps.

The first, and most interesting part of the process is to create new passwords. Password construction rules for special access accounts at NAS are: passwords must be at least eight characters long; passwords must have mixed case, at least one decimal number and they must have one special character. The password construction rules apply to the final password after the password algorithm has been applied. Therefore, the passwords listed on the password sheet may not conform to all of the above construction rules. An example of a NAS password, as listed on the password sheet, is "freem3" and the actual password, after applying the algorithm, is "frEem3-."

Once the passwords are created, the special access database is updated and the new password sheets are printed. The password sheets must either be printed in a secure area or on a printer that someone is monitoring during the entire print job. After all password sheets have been printed and cut to size, a notification is sent, via E-mail to all people with special access. The notification must be sent out a minimum of eight hours in advance and preferably 24 hours before the actual password change occurs.

After the notification has been sent out, people may pick up their new password sheets which are available in the Control Room. All local support personnel are required to sign for their password sheet and may only pick up the sheet belonging to them. Personnel who provide support from a remote site have a local point-of-contact person who picks up and signs for their password sheet. The point-of-contact will then provide the password over the phone, to the remote support personnel, but only when needed. Members of the control room staff are responsible for ensuring that each support person only picks up and signs for his/her password sheet.

The final stage of the task is to manually change the passwords on each system. Passwords are always changed during the second and third shift of the day. A checklist is provided to the people responsible for actual password changes and as each password is changed on a particular system, the person changing the password initializes the checklist for that system. All systems where passwords were not changed during the first pass, due to some problem such as a down system, are retried the next morning. If there are still systems where the password could not be changed for one reason or another, the problem is passed off to the LSA for that system.

3.7 Dealing with Special Access Problems

Just as with any situation where you have a large number of objects and a large group of people interacting with those objects, there are bound to be problems. Special access on NAS systems is no exception. Although NAS has experienced very few problems concerning special access passwords, problems do occur on occasion and need to be addressed. The most common problem seems to be a lost password sheet. Lost password sheets are supposed to be reported to the security analyst within 24 hours. If the password sheet was lost in the vicinity of Ames (i.e., where someone is likely to recognize the system names), then the passwords will be changed that same day. Otherwise, if the sheet was lost in a "relatively" benign area or sent through the washer in a shirt pocket, a new sheet will be printed for the person who lost it and the passwords will remain unchanged.

Another problem concerning special access passwords is the termination (friendly or otherwise) of an employee who has special access. If the person with special access is leaving NAS on a friendly basis, the password will be changed within five working days, or during the next regular change of passwords. However, if the person is leaving under unfriendly circumstances, all passwords and the algorithm will be changed that same day. Major violations of special access by a given individual may also result in changing all passwords for which the individual had access. A major violation would be something along the lines of deliberately destroying system or user files.

4.0 Other Issues

4.1 The Use of Root /.rhost Files

In general, the use of *root /.rhost* files at NAS is not recommended or even allowed. However, there are several justifiable uses for the *root /.rhost* file. One such area is on the NAS workstations. There are currently over 150 workstations at NAS of different architectures and/or operating system revisions. One of the main tools used by the workstation support group to maintain these systems is the **rdist (1)** command. In order to use **rdist (1)** effectively, the NAS workstations must trust *root* "r-commands" from a few selected workstations which serve as binary baseline systems [4]. However, in 98% of the cases, the trust is not bi-directional (i.e., System A may trust *root* from System B, but not vice versa).

Another possible justification for the use of *root* ***/.rhost*** files would be for remote boots of diskless workstations over the network. In general, NAS does not support diskless workstations; however, there are a few occasions when a system must be remotely booted over the network. Once the system has been booted, the ***/.rhost*** entry for the boot server would be removed. Root ***/.rhosts*** files have also been used on the VAX systems to allow remote dumps when a tape drive is down. However, once the tape drive is restored, the ***/.rhosts*** entry will be removed.

4.2 LSU - An alternative to sharing the *root* password

The use of non-unique *root* passwords does present a small problem from time to time. Some users or support personnel will need access to the *root* account on their personal workstation which may be part of a password group. However, it may be the case that management does not wish to provide the individual with *root* access to all workstations in the password group. As an alternative, such users are given *root* access on their systems via the **lsu (1L)** command. Lsu is a a local super-user program written by Matt Bishop [5]. However, I must state that Matt does not recommend using the **lsu (1L)** command to provide *root* access for people.

The **lsu (1L)** command works much in the same manner as the **su (1)** command; however, the password expected is that of the invoking account and not the target account. A configuration file must specify who can use the **lsu (1L)** command and what accounts they can switch to. Use of the **lsu (1L)** command can be restricted to time of day, day of week, month of year and terminal line or dial-up line.

Lsu access is not given out freely. People requesting *root* access, via **lsu (1L)** on a single system must have a good justification for needing the access and they must possess enough knowledge of the UNIX operating system to know what to do and what not to do. Weekly audits are run on the **lsu** configuration files to ensure the files have not been modified. Any system which allows **lsu** access to one or more persons cannot be listed in a *root* ***/.rhosts*** file on any NAS system. In addition to the restriction of ***/.rhosts*** files, the password files on all systems are audited nightly for changes using the **getall (8 L)** command [6], so any change made to a password file will be reported the next day. To date, there have been VERY few problems with using the **lsu (1L)** command to provide *root* access on individual workstations.

5.0 The Future of Special Access Password Security

There are several things that could be done to improve the current level of password security for special access accounts at NAS. The biggest gains could be realized by automating the process of changing passwords. The task of manually changing two or more special access passwords on 180 plus systems is very time consuming and error prone. Ideally, NAS would like to have a secure network administration tool that could change passwords on multiple systems without sending the clear text password over the network. If such a tool were available, the processes to change passwords could be batched and verified ahead of time and then "let loose" when the right time approached. Another area of improvement would be the automatic generation or assignment of passwords. For example, a database could be populated with a year's supply of passwords. At each password change, the new passwords could automatically be assigned to each password group. Reducing the number of password groups would also provide some relief from having to remember numerous passwords; however, this scenario is not likely to be achieved at NAS. Another, perhaps more drastic, means of increasing special access password security would be to reduce the level of privileges for people with special access.

6.0 Summary

This paper has presented the methods used for password security and control at the NAS facility, which can be viewed as a typical large distributed computing environment. Due to the number of systems and the nature of services provided by the NAS facility, password security and control has been a challenging task. The current method which involves the use of password groups, minimum levels of access, a MacIntosh database and printed password sheets has served NAS well over the past several years. However, as the number of NAS systems continues to increase, improvements such as an network password changing utility or even a whole new method, will need to be implemented.

Acknowledgments: Thanks to Bob Van Cleef and Bill Wall for their helpful reviews of the earlier drafts of this paper.

References

[1] Grampp, F.T. and Morris, R.H., "Unix Operating System Security," AT&T Bell Laboratories Technical Journal, 63, 8 (October 1984) 1649 - 1672.

[2] Baldwin, R. W. "Crypt Breakers Workbench," *baldwin@xx.lcs.mit.edu*, October 1986, USENET posting.

[3] Anderson, L. E., "UNIX Password Security," In Proceedings of the USENIX Security Workshop, August 1988, p 7.

[4] Van Cleef, R. E., "System Administration and Maintenance of Fully Configured Workstations," In Proceedings of the USENIX System Administration of Large Scale Installations Conference, November 1988.

[5] Bishop, M., "Collaboration Using Roles," Software -- Practice and Experience, vol. 20, no. 5, pp. 485-497 (May 1990).

[6] Van Cleef, R. E., *getall.* NASA, NASA Ames Research Center, Moffett Field, CA., March 1990, private communications.

6.0 Summary

This paper has presented the methods used for password security and control in the NAS facility, which can be viewed as a typical large distributed computing environment. Due to the number of systems and the number of users accessing the NAS facility, password security and control has been a challenging task. The current method, which involves the use of password groups, minimum levels of access, a centralized database and private password checks has served NAS well over the past several years. However, as the number of NAS systems continues to increase, improvements such as a network password changing utility or even a whole new method will need to be implemented.

Acknowledgments. Thanks to Bob Van Cleef and Bill Weil for their helpful reviews of the earlier drafts of this paper.

References

[1] Grampp, F.T. and Morris, R.H., "Unix Operating System Security", AT&T Bell Laboratories Technical Journal, 63, 8 (October 1984), 1649-1672.

[2] Baldwin, R. W., Crypt Breakers Workbench, [illegible] Massachusetts, October 1986, USENET posting.

[3] Alexander, L. E., "Unix Password Security", In Proceedings of the USENIX Security Workshop, August 1988, [illegible]

[4] Van Cleef, R. E., "System Administration and Maintenance of Fully Configured Workstations," In Proceedings of the USENIX System Administration of Large Scale Installations Conference, November 1988.

[5] Bishop, M., "[illegible]", Software—Practice and Experience, vol. 20, no. 5, pp. [illegible] (May 1990)

[6] Van Cleef, R. E., NASA, NASA-Ames Research Center, Moffett Field, CA, March 1990, private communications.

An Automatic Policy Checker for Controlling Undesirable Program Behaviors

Maria King
UCLA Computer Science Department

May 17, 1990

This paper discusses a new mechanism for comparing selected program properties against a policy, or set of rules, that states allowable program behavior. The motivation for this work is the increased need to control undesirable behaviors of programs, such as those inherent in Trojan horses and computer viruses. This mechanism, called an Automatic Policy Checker (APC), is currently implemented under SunOS[1]. The idea is to explicitly state a system's policy regarding allowable program activity. Subsequently, the APC is used to compare a selected program property against the policy, prior to installation. The APC determines whether a program's specified actions fall within the perimeter of a particular policy.

The term *specification* when applied to programs is usually taken to mean a general statement of *all* of the functional and/or other relevant properties of a program. To distinguish this form of specification from the more general use, the term *mini-spec* is used.

Definition 1 *A mini-spec formally states a selected subset of the functional properties of a program's behavior.*

Definition 2 *A policy is a set of rules that formally states allowable program behavior, in a particular system.*

A formal language, based on regular expressions, has been developed and is used to express both the mini-spec and the policy. An important feature of the APC is that it does not implement any specific policy. Instead, the APC allows experiments with policies intended to prohibit a variety of undesirable program behaviors.

[1] SunOS is a trademark of Sun Microsystems, Incorporated.

The APC has recently been applied to the virus problem[2]. The distinguishing characteristic of a computer virus is its ability to infect other programs by *modifying* them to include a copy of the virus[1]. Experiments for controlling viral activity have focused on the "modification of files" property of programs. Although this property is important to control for virus prevention, the empirical results indicate that policies based on this property result in too many false positives (i.e., too many non-viral programs falling outside the perimeter of the policy and thus being rejected by the APC).

I will discuss my conclusions concerning anti-viral policy in light of the test results. Eleven anti-viral policies were developed as part of the test suite. Twenty-three mini-specs were written to represent 125 Unix[2] programs and the APC was used to test these mini-specs against the eleven policies. Several mini-specs were included to represent programs infected with a virus to determine which policies correctly rejected the infected programs. I will also discuss policies based on other program properties, such as system call patterns and manipulation of environment variables, and I will raise the issue of extending this scheme to include a representation of user behaviors. Lastly, I will propose an implementation of the APC as a run-time tool. It is expected that a run-time APC will have a lower rate of false positives.

References

[1] F. Cohen. Computer Viruses. In *Proceedings of the 7th National Computer Security Conference*, pages 240–263, 1984.

[2] M.M. Pozzo. *Towards Computer Virus Prevention.* PhD thesis, University of California, Los Angeles, 1990.

[2] Unix is a trademark of AT&T.

John Linn
Secure Systems
Digital Equipment Corporation
85 Swanson Road, BXB1-2/D04
Boxborough, MA 01719-1326
Linn@ultra.enet.dec.com; ULTRA::LINN

Generic Security Service Application Program Interface

1 GSSAPI Characteristics and Concepts

This Generic Security Service Application Program Interface (GSSAPI) definition provides security services to calling applications in a generic fashion, supportable with a range of underlying mechanisms and technologies and hence allowing portability of applications to different environments. It separates the operations of initializing a security context between peers, achieving peer entity authentication[1] (Init_sec_context( ) and Accept_sec_context( ) calls), from the operations of providing per-message data origin authentication and data integrity protection (Sign( ) and Verify( ) calls) for messages subsequently transferred in that context. Per-message Seal( ) and Unseal( ) calls provide the data origin authentication and data integrity services which Sign( ) and Verify( ) offer, and also support selection of confidentiality services as a caller option.

The GSSAPI design assumes and addresses several basic goals, including:

- Mechanism independence: The GSSAPI defines an interface to security services at a generic level which is independent of particular underlying mechanisms. For example, GSSAPI-provided services can be implemented by secret-key technologies (e.g., Kerberos) or public-key approaches (e.g., X.509).
- Protocol environment independence: The GSSAPI is independent of the communications protocol suites with which it is employed, permitting use in a broad range of protocol environments. In appropriate environments, an intermediate implementation "veneer" which is oriented to a particular communication protocol (e.g., RPC or ACSE) may be interposed between applications and the GSSAPI, invoking GSSAPI facilities in conjunction with the selected protocol.
- Protocol association independence: The GSSAPI's security context construct is independent of communications protocol association constructs. This characteristic allows a single GSSAPI implementation to be utilized by a variety of invoking protocol modules on behalf of those modules' calling applications. GSSAPI services can also be invoked directly by applications, wholly independent of protocol associations.
- Suitability to a range of implementation placements: GSSAPI clients are not constrained to reside within any Trusted Computing Base (TCB) perimeter defined on a system where the GSSAPI is implemented; security services are specified in a manner suitable to both intra-TCB and extra-TCB callers.

1.1 GSSAPI Constructs

This section describes basic elements comprising the GSSAPI.

[1] This security service definition, and other definitions used in this document, corresponds to that provided in International Standard ISO 7498-2-1988(E), Security Architecture.

1.1.1 Credentials

Credentials structures provide the prerequisites enabling peers to establish security contexts with each other. GSSAPI callers reference credentials structures indirectly, through GSSAPI-provided credential handles ("cred_handles"). A single credentials structure may accomodate credentials for multiple mech_types; a credentials structure's contents will vary depending on the set of supported mech_types.

It is the responsibility of underlying system-specific mechanisms and OS functions below the GSSAPI to ensure that the ability to acquire and use credentials associated with a given identity is constrained to appropriate processes within a system. This responsibility should be taken seriously by implementors, as the ability for an entity to utilize a principal's credentials is equivalent to the entity's ability to successfully assert that principal's identity.

Once a set of GSSAPI credentials is established, the transferability of that credentials set to other processes or analogous constructs within a system is a local matter, not defined by the GSSAPI. An example local policy would be one in which any credentials received as a result of login to a given user account, or of delegation of rights to that account, are accessible by, or transferable to, processes running under that account.

The credential establishment process (particularly when performed on behalf of users rather than server processes) is likely to require access to passwords or other quantities which should be protected locally and exposed for the shortest time possible. As a result, it may often be appropriate for preliminary credential establishment to be performed through local means in conjunction with login. The resulting preliminary credentials would be set aside for subsequent acquisition by the GSSAPI Acquire_cred() call.

Means for selection of option values which are bound into credentials and which are specific to particular mech_types (e.g., certain Kerberos V5 TGT options) are local matters, and will likely be based on selectively user-overridable defaults established as a system management function. The period for which an established credentials set remains valid is also a local matter, and relates in part to a tradeoff between revocation responsiveness and a desire to avoid disrupting users in the course of ordinary interactive sessions.

1.1.2 Security Contexts

Security contexts are established between peers, using credentials established locally in conjunction with each peer or received by peers via delegation. Multiple contexts may exist simultaneously between a pair of peers, using the same or different sets of credentials. Coexistence of multiple contexts using different credentials allows graceful rollover when credentials expire. Distinction among multiple contexts based on the same credentials serves applications by distinguishing different message streams in a security sense.

The GSSAPI is independent of underlying protocols and addressing structure, and depends on its callers to transport GSSAPI-provided data elements. As a result of these factors, it is a caller responsibility to parse communicated messages, separating GSSAPI-related data elements from caller-provided data. The GSSAPI is independent of connection vs. connectionless orientation of the underlying communications service.

No correlation between security context and communications protocol association is dictated. This separation allows the GSSAPI to be used in a wide range of communications environments, and also simplifies the calling sequences of the individual calls. In many cases (depending on underlying security protocol, associated mechanism, and availability of cached information), the state information required for context setup can be sent concurrently with initial signed user data, without interposing additional message exchanges.

1.1.3 Mechanism Types

In order to initiate a security context with a target peer, it is necessary to determine an appropriate underlying mechanism type (mech_type) which is shared with that peer. In some cases, mech_type determinations may be made through syntactic convention by examining a target name; in other cases, a database or naming service attribute lookup may be needed in order to provide mech_type information corresponding to a provided name.

It is recommended that callers initiating contexts use a default mech_type value, allowing system-specific functions below the GSSAPI to select the appropriate mech_type, but callers may direct that a particular mech_type be employed when necessary. The set of mech_types which an entity may assert as an initiator in establishing contexts to others need not be the same as the set of mech_types with which it can accept incoming contexts from others.

A registry of mech_type identifiers is needed in order to preclude ambiguous interpretation. This document does not enumerate the set of mech_types which may support GSSAPI functions. Kerberos (V5) and X.509 exemplify classes of mechanisms and associated mech_types; it is possible, however, that multiple mech_types would be appropriate within such classes in order to distinguish protocol alternatives (e.g., Kerberos single-TGT vs. double-TGT).

1.1.4 Channel Bindings

The GSSAPI accommodates the concept of caller-provided channel binding ("chan_binding") information, used by GSSAPI callers to bind a security context to relevant characteristics (e.g., addresses, transformed representations of encryption keys) of the underlying communications channel and of protection mechanisms applied to that communications channel. Verification by one peer of chan_binding information provided by the other peer to a context serves to protect against various active attacks. The caller initiating a security context must determine the chan_binding values before making the Init_sec_context() call. The appropriate form for channel binding information is dependent on the communications protocol environment in which the GSSAPI is used, and so is not defined in this specification.

1.2 GSSAPI Features and Issues

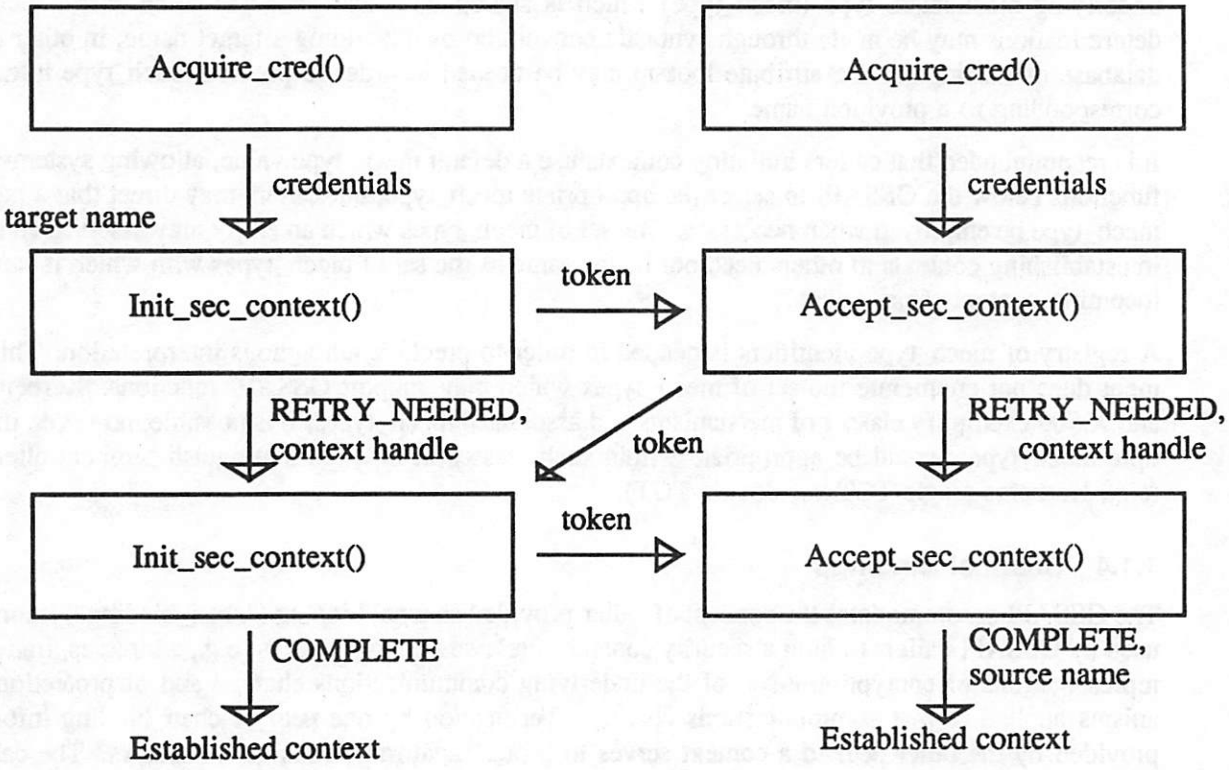
This section describes aspects of GSSAPI operations, of the security services which the GSSAPI provides, and provides commentary on design issues.

1.2.1 Status Reporting

Each GSSAPI call provides two status returns: major_status represents high-level information about the completion of a call (e.g., COMPLETE, FAILURE, RETRY_NEEDED), sufficient to drive normal control flow within the caller in a generic fashion, and minor_status provides more detailed status information which may include status codes specific to the underlying security mechanism. Minor_status values are not specified in this document.

RETRY_NEEDED major_status returns, and optional message outputs, are provided in Init_sec_context() and Accept_sec_context() calls so that invocations of multiple message preamble transactions (as are required, for example, to authenticate to a Kerberos V5 double-TGT service) need not be reflected in separate code paths within calling applications. The same mechanism is used to encapsulate mutual authentication within the GSSAPI's context initiation calls. Figure 1 illustrates a GSSAPI retry scenario.

Figure 1: Example Context Establishment with Retry



For mech_types which require interactions with third-party servers in order to establish a security context, GSSAPI context establishment calls may block pending completion of such third-party interactions. On the other hand, no GSSAPI calls pend on serialized interactions with GSSAPI peer entities. As a result, local GSSAPI status returns cannot reflect unpredictable or asynchronous exceptions occurring at remote peers, and reflection of such status information is a caller responsibility outside the GSSAPI.

1.2.2 Per-Message Replay Detection and Sequencing

Certain underlying mech_types are expected to offer support for replay detection and/or sequencing of messages transferred on the contexts they support. Such optionally-selectable, per-message features are different from replay detection and sequencing features applied to the context establishment operation itself; the presence or absence of context-level replay or sequencing features is wholly a function of the underlying mech_type's capabilities, and is not selected or omitted as a caller option.

The caller initiating a context provides flags (replay_det_req_flag and sequence_req_flag) to specify whether the use of per-message replay detection and sequencing features is desired on the context being established. The GSSAPI implementation at the initiator system can determine whether these features are supported (and whether they are optionally selectable) as a function of mech_type, without need for bilateral negotiation with the target. When enabled, these features provide recipients with indicators as a result of GSSAPI processing of incoming messages, identifying whether those messages were detected as duplicates or out-of-sequence. Detection of such events does not prevent a suspect message from being provided to a recipient; the appropriate course of action on a suspect message is a matter of caller policy.

The semantics of the replay detection and sequencing services applied to received messages, as visible across the interface which the GSSAPI provides to its clients, are as follows:

When replay_det_state is TRUE, the possible major_status returns for well-formed and correctly signed messages are as follows:

1. COMPLETE indicates that the message was within the window (of time or sequence space) allowing replay events to be detected, and that the message was not a replay of a previously-processed message within that window.
2. DUPLICATE_TOKEN indicates that the signature on the received message was correct, but that the message was recognized as a duplicate of a previously-processed message.
3. OLD_TOKEN indicates that the signature on the received message was correct, but that the message is too old to be checked for duplication.

When sequence_state is TRUE, the possible major_status returns for well-formed and correctly signed messages are as follows:

1. COMPLETE indicates that the message was within the window (of time or sequence space) allowing replay events to be detected, and that the message was not a replay of a previously-processed message within that window.
2. DUPLICATE_TOKEN indicates that the signature on the received message was correct, but that the message was recognized as a duplicate of a previously-processed message.
3. OLD_TOKEN indicates that the signature on the received message was correct, but that the token is too old to be checked for duplication.
4. UNSEQ_TOKEN indicates that the signature on the received message was correct, but that it is earlier in a sequenced stream [2] than a message already processed on the context.

As the message stream integrity features (especially sequencing) may interfere with certain applications' intended communications paradigms, and since support for such features is likely to be resource intensive, it is highly recommended that mech_types supporting these features allow them to be activated selectively on initiator request when a context is established. A context initiator and target are provided with corresponding indicators (replay_det_state and sequence_state), signifying whether these features are active on a given context.

An example mech_type [3] supporting per-message replay detection might (when replay_det_state is TRUE) implement the feature as follows: The underlying mechanism would insert timestamps in data elements output by Sign() and Seal(), and would maintain (within a time-limited window) a cache (qualified by originator-recipient pair) identifying received data elements processed by Verify() and Unseal(). When this feature is active, exception status returns (DUPLICATE_TOKEN, OLD_TOKEN) will be provided when Verify() or Unseal() is presented with a message which is either a detected duplicate of a prior message or which is too old to validate against a cache of recently received messages.

[2] Mechanisms can be architected to provide a stricter form of sequencing service, delivering particular messages to recipients only after all predecessor messages in an ordered stream have been delivered. This type of support is incompatible with the GSSAPI paradigm in which recipients receive all messages, whether in order or not, and provide them (one at a time, without intra-GSSAPI message buffering) to GSSAPI routines for validation. GSSAPI facilities provide supportive functions, aiding clients to achieve strict message stream integrity in an efficient manner in conjunction with sequencing provisions in communications protocols, but the GSSAPI does not offer this level of message stream integrity service by itself.

[3] The example mechanism described here closely corresponds to that described in the Kerberos V5 (Draft 2, 6 November 1989) specification, Section 2.2.3.

1.2.3 Quality of Protection

Some mech_types will provide their users with fine granularity control over the means used to provide per-message protection, allowing callers to trade off security processing overhead dynamically against the protection requirements of particular messages. A per-message quality-of-protection parameter (analogous to quality-of-service, or QOS) selects (in a mechanism-specific way) among different QOP options supported by that mechanism. Callers not wishing to exert explicit mechanism-specific QOP control may request selection of a default QOP. On context establishment for a multi-QOP mech_type, context-level data provides the prerequisite data for a range of protection qualities.

1.2.4 Naming

As far as the GSSAPI is concerned, authenticated names are opaque strings with no structure dictated. It must be recognized, however, that underlying security mechanisms will impose constraints on naming representations. Translation between different naming representations is a function outside the scope of this interface, and support by local OS-specific functions is anticipated to map between application-visible forms and conventions of underlying security mechanisms.

Use of a common hierarchic naming format would aid in development of portable applications. Selection or specification of such a format is a key issue for support of heterogeneous distributed computing, with pervasive impact significantly broader than security concerns. We recommend that this general naming issue be addressed in other, more appropriate forums, and exclude its resolution from the scope of the GSSAPI.

2 Interface Descriptions

This section describes the GSSAPI's service interface, dividing the set of calls offered into three groups. Credential management calls are related to the acquisition and release of credentials by principals. Context-level calls are related to the management of security contexts between principals. Per-message calls are related to the protection of individual messages on established security contexts. Table 1 groups and summarizes the calls in tabular fashion.

Table 1: GSSAPI Calls

CREDENTIAL MANAGEMENT	
Acquire_cred	acquire credentials for use
Release_cred	release credentials after use
CONTEXT-LEVEL CALLS	
Init_sec_context	initiate outbound security context
Accept_sec_context	accept inbound security context
Delete_sec_context	flush context when no longer needed
Process_context_token	process received control token on context
PER-MESSAGE CALLS	
Sign	apply signature, receive as token separate from message
Verify	validate signature token along with message
Seal	sign, optionally encrypt, encapsulate
Unseal	decapsulate, decrypt if needed, validate signature

2.1 Credential management calls

These GSSAPI calls provide functions related to the management of credentials. Their characterization with regard to whether or not they may block pending exchanges with other network entities (e.g., directories or authentication servers) depends in part on OS-specific (extra-GSSAPI) issues, so is not specified in this document.

The Acquire_cred( ) call is defined within the GSSAPI in support of application portability, with a particular orientation towards support of portable server applications. OS-specific means to acquire credentials for use with the GSSAPI may also exist, for example in conjunction with OS-specific login functions based on passwords or other user authentication technologies. The Release_cred( ) call provides a means for callers to indicate to the GSSAPI that use of a credentials structure is no longer required.

2.1.1 Acquire_cred call

Inputs:

- desiredname OCTET STRING,
- time_req INTEGER,—in seconds; 0 requests default
- desired_mechs SET OF INTEGER—empty set requests system-selected default

Outputs:

- major_status INTEGER,
- minor_status INTEGER,
- output_cred_handle OCTET STRING,
- time_rec INTEGER —in seconds

Return major_status values:

- COMPLETE indicates that requested credentials were successfully established, for the requested duration and for the requested mech_types, and that those credentials can be referenced for subsequent use with the handle returned in output_cred_handle.
- BAD_MECH indicates that an unavailable mech_type was requested, causing the credential establishment operation to fail.
- FAILURE indicates that credential establishment failed for other reasons, including lack of authorization to establish and use credentials associated with the identity named in the input desiredname argument.

Acquire_cred( ) is used to establish credentials so that a principal can initiate and accept security contexts under the identity represented by the desiredname input argument. On successful completion, the returned output_cred_handle result provides a handle for subsequent references to the established credentials. The time_rec result indicates the length of time for which the acquired credentials will be valid, as an offset from the present.

The caller of Acquire_cred( ) can designate the length of time for which established credentials are to be valid (time_req argument), beginning at the present[4], or can request credentials with a default validity interval. The caller can explicitly specify a set of mech_types which are to be accomodated in the returned credentials (desired_mechs argument), or can request credentials for a system-defined default set of mech_types.

[4] Requests for postdated credentials are not supported within the GSSAPI.

2.1.2 Release_cred call

Input:

- cred_handle OCTET STRING

Outputs:

- major_status INTEGER,
- minor_status INTEGER

Return major_status values:

- COMPLETE indicates that the credentials referenced by the input cred_handle were deleted for purposes of subsequent access by the caller. The effect on other processes which may be authorized shared access to such credentials is a local matter.
- NO_CRED indicates that no deletion operation was performed, either because the input cred_handle was invalid or because the caller lacks authorization to access the referenced credentials.

Provides a means for a caller to explicitly request that credentials be deleted when their use is no longer required. Note that system-specific credential management functions are also likely to exist, for example to assure that credentials shared among processes are properly deleted when all affected processes terminate, even if no explicit deletion requests are issued by those processes.

2.2 Context-level calls

This group of calls is devoted to the establishment and management of security contexts between peers. A context's initiator calls Init_sec_context(), resulting in generation of a token which the caller passes to the target. At the target, that token is passed to Accept_sec_context(). Depending on the underlying mech_type and specified options, additional token exchanges may be performed in the course of context establishment; such exchanges are accomodated by RETRY_NEEDED status returns from Init_sec_context() and Accept_sec_context(). Either party to an established context may invoke Delete_sec_context() to flush context information when a context is no longer required. Process_context_token() is used to process received tokens carrying context-level control information.

2.2.1 Init_sec_context call

Inputs:

- claimant_cred_handle OCTET STRING, —NULL specifies "use default"
- input_context_handle INTEGER, —0 specifies "none assigned yet"
- targname OCTET STRING,
- mech_type INTEGER, —0 specifies "use default"
- deleg_req_flag BOOLEAN,
- mutual_req_flag BOOLEAN,
- replay_det_req_flag BOOLEAN,
- sequence_req_flag BOOLEAN,
- chan_bindings OCTET STRING,

- input_token OCTET STRING

Outputs:

- major_status INTEGER,
- minor_status INTEGER,
- output_context_handle INTEGER,
- output_token OCTET STRING,
- deleg_state BOOLEAN,
- mutual_state BOOLEAN,
- replay_det_state BOOLEAN,
- sequence_state BOOLEAN

This call may block pending network interactions for those mech_types in which an authentication server or other network entity must be consulted on behalf of a context initiator in order to generate an output_token suitable for presentation to a specified target.

Return major_status values:

- COMPLETE indicates that context-level information was successfully initialized, and that the returned output_token will provide sufficient information for the target to perform per-message processing on the newly-established context.
- DEFECTIVE_TOKEN indicates that consistency checks performed on the input_token failed, preventing further processing from being performed based on that token.
- RETRY_NEEDED indicates that control information in the returned output_token must be sent to the target, and that a reply must be received and passed as the input_token argument to a retry call to Init_sec_context( ), before per-message processing can be performed in conjunction with this context.
- CREDENTIALS_EXPIRED indicates that the credentials provided through the input claimant_cred_handle argument are no longer valid, so context establishment cannot be completed.
- BAD_BINDINGS indicates that a mismatch between the caller-provided chan_bindings and those extracted from the input_token was detected, signifying a security-relevant event and preventing context establishment. (This result will be returned by Init_sec_context only for contexts where mutual_state is TRUE.)
- NO_CONTEXT indicates that no valid context was recognized for the input context_handle provided; this major status will be returned only for successor calls following RETRY_NEEDED status returns.
- FAILURE indicates that context setup could not be accomplished, and that no interface-defined recovery action is available.

Used by context initiator, providing an output_token suitable for use by the target within the selected mech_type's protocol. Using information in the credentials structure referenced by claimant_cred_handle, initialize the data structures required to establish a security context with target targname. The claimant_cred_handle must correspond to the same valid credentials structure on the initial call to Init_sec_context( ) and on any successor calls resulting from RETRY_NEEDED status returns; different protocol sequences modeled by the RETRY_NEEDED mechanism will require access to credentials at different points in the context establishment sequence.

The input_context_handle argument is 0, specifying "not yet assigned", on the first Init_sec_context() call relating to a given context. That call returns an output_context_handle for future references to this context. When retry attempts to Init_sec_context() are needed to perform context establishment, the previously-returned non-zero handle value is entered into the input_context_handle argument and will be echoed in the returned output_context_handle argument.

The chan_bindings argument is used by the caller to provide information binding the security context to security-related characteristics (e.g., addresses, cryptographic keys) of the underlying communications channel. The input_token argument contains a message received from the target, and is significant only on a call to Init_sec_context() which follows a previous return indicating RETRY_NEEDED major_status.

It is the caller's responsibility to establish a communications path to the target, and to transmit the output_token to the target over that path. The output_token can, however, be transmitted along with the first application-provided input message processed by Sign() or Seal() in conjunction with this context.

The initiator may request various context-level functions through input flags: the deleg_req_flag requests delegation of access rights, the mutual_req_flag requests mutual authentication, the replay_det_req_flag requests that replay detection features be applied to messages transferred on the established context, and the sequence_req_flag requests that sequencing be enforced. (See Section 1.2.2 for more information on replay detection and sequencing features.) Not all of these features will be available in all underlying mech_types; the corresponding return state values indicate, as a function of mech_type processing capabilities and initiator-provided input flags, the set of features which will be active on the context. Failure to provide the precise set of features requested by the caller does not cause context establishment to fail; it is the caller's prerogative to delete the context if the feature set provided is unsuitable for the caller's use.

If the mutual_state is TRUE, this fact will be reflected in the output_token. A call to Accept_sec_context() at the target in conjunction with such a context will return a token, to be processed by a retry call to Init_sec_context(), in order to achieve mutual authentication.

2.2.2 Accept_sec_context call

Inputs:

- acceptor_cred_handle OCTET STRING,
- input_context_handle INTEGER, —0 specifies "not yet assigned"
- chan_bindings OCTET STRING,
- input_token OCTET STRING

Outputs:

- major_status INTEGER,
- minor_status INTEGER,
- srcname OCTET STRING,
- mech_type INTEGER,
- output_context_handle INTEGER,
- deleg_state BOOLEAN,
- mutual_state BOOLEAN,
- replay_det_state BOOLEAN,

- sequence_state BOOLEAN,
- delegated_cred_handle OCTET STRING,
- output_token OCTET STRING

This call may block pending network interactions for those mech_types in which a directory service or other network entity must be consulted on behalf of a context acceptor in order to validate a received input_token.

Return major_status values:

- COMPLETE indicates that context-level data structures were successfully initialized, and that per-message processing can now be performed in conjunction with this context.
- DEFECTIVE_TOKEN indicates that consistency checks performed on the input_token failed, preventing further processing from being performed based on that token.
- RETRY_NEEDED indicates that control information in the returned output_token must be sent to the initiator, and that a response must be received and passed as the input_token argument to a retry call to Accept_sec_context( ), before per-message processing can be performed in conjunction with this context.
- BAD_SIG indicates that the received input_token contains an incorrect signature, so context setup cannot be accomplished.
- DUPLICATE_TOKEN indicates that the signature on the received input_token was correct, but that the input_token was recognized as a duplicate of an input_token already processed. No new context is established.
- OLD_TOKEN indicates that the signature on the received input_token was correct, but that the input_token is too old to be checked for duplication against previously-processed input_tokens. No new context is established.
- CREDENTIALS_EXPIRED indicates that the credentials provided through the input acceptor_cred_handle argument are no longer valid, so context establishment cannot be completed.
- BAD_BINDINGS indicates that a mismatch between the caller-provided chan_bindings and those extracted from the input_token was detected, signifying a security-relevant event and preventing context establishment.
- NO_CONTEXT indicates that no valid context was recognized for the input context_handle provided; this major status will be returned only for successor calls following RETRY_NEEDED status returns.
- FAILURE indicates that context setup could not be accomplished, and that no interface-defined recovery action is available.

Used by context target. Using information in the credentials structure referenced by the input acceptor_cred_handle, verify the incoming input_token and (assuming success) return the authenticated srcname and the mech_type used. The acceptor_cred_handle must correspond to the same valid credentials structure on the initial call to Accept_sec_context( ) and on any successor calls resulting from RETRY_NEEDED status returns; different protocol sequences modeled by the RETRY_NEEDED mechanism will require access to credentials at different points in the context establishment sequence.

The input_context_handle argument is 0, specifying "not yet assigned", on the first Accept_sec_context( ) call relating to a given context. That call returns an output_context_handle for future references to this context; when retry attempts to Accept_sec_context( ) are needed to perform context establishment, that handle value will be entered into the input_context_handle argument.

The chan_bindings argument is used by the caller to provide information binding the security context to security-related characteristics (e.g., addresses, cryptographic keys) of the underlying communications channel.

The returned state results (deleg_state, mutual_state, replay_det_state, and sequence_state) reflect the same context state values as returned to Init_sec_context()'s caller at the initiator system.

The delegated_cred_handle result is significant only when deleg_state is TRUE, and provides a means for the target to reference the delegated credentials. The output_token result, when non-NULL, provides a control message to be returned to the initiator.

Note: A target must be able to distinguish a context-level input_token, which is passed to Accept_sec_context(), from the per-message data elements passed to Verify() or Unseal(). These data elements may arrive in a single application message, and Accept_sec_context() must be performed before per-message processing can be performed successfully.

2.2.3 Delete_sec_context call

Input:

- context_handle INTEGER

Outputs:

- major_status INTEGER,
- minor_status INTEGER,
- output_context_token OCTET STRING

Return major_status values:

- COMPLETE indicates that the context was recognized, that relevant context-specific information was flushed, and that the returned output_context_token is ready for transfer to the context's peer.
- FAILURE indicates that the context is recognized, but that the Delete_sec_context() operation could not be performed for reasons identified by minor_status codes.
- NO_CONTEXT indicates that no valid context was recognized for the input context_handle provided.

This call may block pending network interactions for mech_types in which active notification must be made to a central server when a security context is to be deleted.

This call can be made by either peer in a security context, to flush context-specific information and to return an output_context_token which can be passed to the context's peer informing it that the peer's corresponding context information can also be flushed. (Once a context is established, the peers involved are expected to retain cached credential and context-related information until the information's expiration time is reached or until a Delete_sec_context() call is made.) Attempts to perform per-message processing on a deleted context will result in error returns.

2.2.4 Process_context_token call

Input:

- context_handle INTEGER,

- input_context_token OCTET STRING

Outputs:

- major_status INTEGER,
- minor_status INTEGER,

Return major_status values:

- COMPLETE indicates that the input_context_token was successfully processed in conjunction with the context referenced by context_handle.
- DEFECTIVE_TOKEN indicates that consistency checks performed on the received context_token failed, preventing further processing from being performed with that token.
- FAILURE indicates that the context is recognized, but that the Process_context_token() operation could not be performed for reasons identified by minor_status codes.
- NO_CONTEXT indicates that no valid context was recognized for the input context_handle provided.

This call is used to process context_tokens received from peers, with corresponding impact on context-level state information. Its only current use is to process the context_tokens generated by Delete_sec_context(), and will not block pending network interactions for that purpose.

2.3 Per-message calls

This group of calls is used to perform per-message protection processing on an established security context. None of these calls block pending network interactions. These calls may be invoked by a context's initiator or by the context's target. The four members of this group should be considered as two pairs; the output from Sign() is properly input to Verify(), and the output from Seal() is properly input to Unseal().

Sign() and Verify() support data origin authentication and data integrity services. When Sign() is invoked on an input message, it yields a token containing data items which allow underlying mechanisms to provide the specified security services. The original message, along with the generated token, is passed to the remote peer; these two data elements are processed by Verify(), which validates the message in conjunction with the separate token.

Seal() and Unseal() support caller-requested confidentiality in addition to the data origin authentication and data integrity services offered by Sign() and Verify(). Seal() outputs a single data element, encapsulating optionally-enciphered user data as well as associated token data items. The data element output from Seal() is passed to the remote peer and processed by Unseal() at that system. Unseal() combines decipherment (as required) with validation of data items related to authentication and integrity.

2.3.1 Sign call

Inputs:

- context_handle INTEGER,
- qop_req INTEGER,—0 specifies default QOP
- message OCTET STRING

Outputs:

- major_status INTEGER,

- minor_status INTEGER,
- per_msg_token OCTET STRING

Return major_status values:

- COMPLETE indicates that a signature, suitable for an established security context, was successfully applied and that the message and corresponding per_msg_token are ready for transmission.
- CONTEXT_EXPIRED indicates that context-related data items have expired, so that the requested operation cannot be performed.
- CREDENTIALS_EXPIRED indicates that the context is recognized, but that its associated credentials have expired, so that the requested operation cannot be performed.
- FAILURE indicates that the context is recognized, but that the requested operation could not be performed for reasons identified by minor_status codes.
- NO_CONTEXT indicates that no valid context was recognized for the input context_handle provided.

Using the security context referenced by context_handle, apply a signature to the input message (along with timestamps and/or other data included in support of mech_type-specific mechanisms) and return the result in per_msg_token. The qop_req parameter allows quality-of-protection control. The caller passes the message and the per_msg_token to the target.

The Sign() function completes before the message and per_msg_token is sent to the peer; successful application of Sign() does not guarantee that a corresponding Verify() has been (or can necessarily be) performed successfully when the message arrives at the destination.

2.3.2 Verify call

Inputs:

- context_handle INTEGER,
- message OCTET STRING,
- per_msg_token OCTET STRING

Outputs:

- qop_state INTEGER,
- major_status INTEGER,
- minor_status INTEGER,

Return major_status values:

- COMPLETE indicates that the message was successfully verified.
- DEFECTIVE_TOKEN indicates that consistency checks performed on the received per_msg_token failed, preventing further processing from being performed with that token.
- BAD_SIG indicates that the received per_msg_token contains an incorrect signature for the message.
- DUPLICATE_TOKEN, OLD_TOKEN, and UNSEQ_TOKEN values appear in conjunction with the optional per-message replay detection features described in Section 1.2.2; their semantics are described in that section.
- CONTEXT_EXPIRED indicates that context-related data items have expired, so that the requested operation cannot be performed.

- CREDENTIALS_EXPIRED indicates that the context is recognized, but that its associated credentials have expired, so that the requested operation cannot be performed.
- FAILURE indicates that the context is recognized, but that the Verify() operation could not be performed for reasons identified by minor_status codes.
- NO_CONTEXT indicates that no valid context was recognized for the input context_handle provided.

Using the security context referenced by context_handle, verify that the input per_msg_token contains an appropriate signature for the input message, and apply any active replay detection or sequencing features. Return an indication of the quality-of-protection applied to the processed message in the qop_state result.

2.3.3 Seal call

Inputs:

- context_handle INTEGER,
- conf_req_flag BOOLEAN,
- qop_req INTEGER,—0 specifies default QOP
- input_message OCTET STRING

Outputs:

- major_status INTEGER,
- minor_status INTEGER,
- conf_state BOOLEAN,
- output_message OCTET STRING

Return major_status values:

- COMPLETE indicates that the input_message was successfully processed and that the output_message is ready for transmission.
- CONTEXT_EXPIRED indicates that context-related data items have expired, so that the requested operation cannot be performed.
- CREDENTIALS_EXPIRED indicates that the context is recognized, but that its associated credentials have expired, so that the requested operation cannot be performed.
- FAILURE indicates that the context is recognized, but that the Seal() operation could not be performed for reasons identified by minor_status codes.
- NO_CONTEXT indicates that no valid context was recognized for the input context_handle provided.

Performs the data origin authentication and data integrity functions of Sign(). If the input conf_req_flag is TRUE, requests that confidentiality be applied to the input_message. Confidentiality may not be supported in all mech_types or by all implementations; the returned conf_state flag indicates whether confidentiality was provided for the input_message. The qop_req parameter allows quality-of-protection control.

In all cases, the Seal() call yields a single output_message data element containing (optionally enciphered) user data as well as control information.

2.3.4 Unseal call

Inputs:

- context_handle INTEGER,
- input_message OCTET STRING

Outputs:

- conf_state BOOLEAN,
- qop_state INTEGER,
- major_status INTEGER,
- minor_status INTEGER,
- output_message OCTET STRING

Return major_status values:

- COMPLETE indicates that the input_message was successfully processed and that the resulting output_message is available.
- DEFECTIVE_TOKEN indicates that consistency checks performed on the per_msg_token extracted from the input_message failed, preventing further processing from being performed.
- BAD_SIG indicates that an incorrect signature was detected for the message.
- DUPLICATE_TOKEN, OLD_TOKEN, and UNSEQ_TOKEN values appear in conjunction with the optional per-message replay detection features described in Section 1.2.2; their semantics are described in that section.
- CONTEXT_EXPIRED indicates that context-related data items have expired, so that the requested operation cannot be performed.
- CREDENTIALS_EXPIRED indicates that the context is recognized, but that its associated credentials have expired, so that the requested operation cannot be performed.
- FAILURE indicates that the context is recognized, but that the Unseal( ) operation could not be performed for reasons identified by minor_status codes.
- NO_CONTEXT indicates that no valid context was recognized for the input context_handle provided.

Processes a data element generated (and optionally enciphered) by Seal( ), provided as input_message. The returned conf_state value indicates whether confidentiality was applied to the input_message. If conf_state is TRUE, Unseal( ) deciphers the input_message. Returns an indication of the quality-of-protection applied to the processed message in the qop_state result. Seal( ) performs the data integrity and data origin authentication checking functions of Verify( ) on the plaintext data. Plaintext data is returned in output_message.

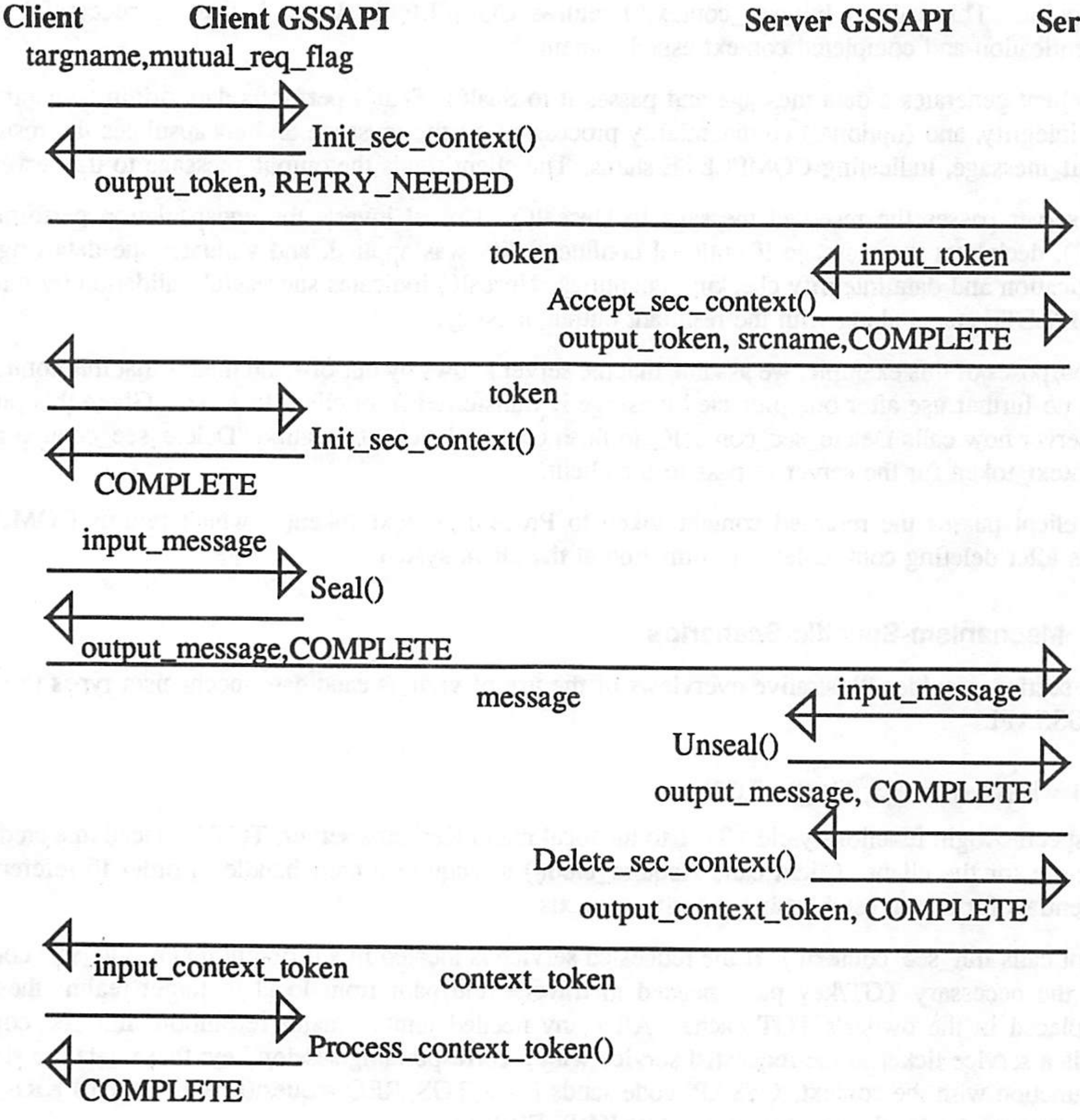
3 Example Scenarios

These discussions are intended as examples for clarification, demonstrating how GSSAPI functions can be used and implemented by candidate underlying mechanisms. They should not be regarded as constrictive to implementations or as defining the only means through which GSSAPI functions can be realized with a particular underlying technology, and do not demonstrate all GSSAPI features with each technology.

3.1 Client-Oriented Scenario

Figure 2 illustrates the dataflows involved in use of the GSSAPI by a client and server in a mechanism-independent fashion, establishing a security context and transferring a protected message. The example assumes that credential acquisition has already been completed. Only a subset of parameter and result values are illustrated, for reasons of clarity in exposition.

Figure 2: Example Client Scenario



The client calls Init_sec_context( ) to establish a security context to the server identified by targname, and elects to set the mutual_req_flag so that mutual authentication is performed in the course of context establishment. Init_sec_context( ) returns an output_token to be passed to the server, and indicates RETRY_NEEDED status pending completion of the mutual authentication sequence. Had mutual_req_ flag not been set, the initial call to Init_sec_context() would have returned COMPLETE status. The client sends the output_token to the server.

The server passes the received token as the input_token parameter to Accept_sec_context(). Accept_sec_context indicates COMPLETE status, provides the client's authenticated identity in the srcname result, and provides an output_token to be passed to the client. The server sends the output_token to the client.

The client passes the received token as the input_token parameter to a successor call to Init_sec_context(), which processes data included in the token in order to achieve mutual authentication from the client's viewpoint. This call to Init_sec_context() returns COMPLETE status, indicating successful mutual authentication and completed context establishment.

The client generates a data message and passes it to Seal(). Seal() performs data origin authentication, data integrity, and (optional) confidentiality processing on the message and encapsulates the result into output_message, indicating COMPLETE status. The client sends the output_message to the server.

The server passes the received message to Unseal(). Unseal inverts the encapsulation performed by Seal(), deciphers the message if optional confidentiality was applied, and validates the data origin authentication and data integrity checking quantities. Unseal() indicates successful validation by returning COMPLETE status along with the resultant output_message.

For purposes of this example, we assume that the server knows by out-of-band means that this context will have no further use after one protected message is transferred from client to server. Given this premise, the server now calls Delete_sec_context() to flush context-level information. Delete_sec_context returns a context_token for the server to pass to the client.

The client passes the returned context_token to Process_context_token(), which returns COMPLETE status after deleting context-level information at the client system.

3.2 Mechanism-Specific Scenarios

This section provides illustrative overviews of the use of various candidate mechanism types to support the GSSAPI.

3.2.1 Kerberos V5, single-TGT

OS-specific login functions yield a TGT to the local realm Kerberos server; TGT is placed in a credentials structure for the client. Client calls Acquire_cred() to acquire a cred_handle in order to reference the credentials for use in establishing security contexts.

Client calls Init_sec_context(). If the requested service is located in a different realm, Init_sec_context() gets the necessary TGT/key pairs needed to traverse the path from local to target realm; these data are placed in the owner's TGT cache. After any needed remote realm resolution, Init_sec_context() yields a service ticket to the requested service with a corresponding session key; these data are stored in conjunction with the context. GSSAPI code sends KRB_TGS_REQ request(s) and receives KRB_TGS_REP response(s) (in the successful case) or KRB_ERROR.

Assuming success, Init_sec_context() builds a Kerberos-formatted KRB_AP_REQ message, and returns it in output_token. The client sends the output_token to the service.

The service passes the received token as the input_token argument to Accept_sec_context(), which verifies the authenticator, provides the service with the client's authenticated name, and returns an output_context_handle.

Both parties now hold the session key associated with the service ticket, and can use this key in subsequent Sign(), Verify(), Seal(), and Unseal() operations.

3.2.2 Kerberos V5, double-TGT

TGT acquisition as above.

Note: To avoid unnecessary frequent invocations of error paths when implementing the GSSAPI atop Kerberos V5, it seems appropriate to represent "single-TGT K-V5" and "double-TGT K-V5" with separate mech_types, and this discussion makes that assumption.

Based on the (specified or defaulted) mech_type, Init_sec_context() determines that the double-TGT protocol should be employed for the specified target. Init_sec_context() returns RETRY_NEEDED major_status[5], and its returned output_token contains a request to the service for the service's TGT. (If a service TGT with suitably long remaining lifetime already exists in a cache, it may be usable, obviating the need for this step.) The client passes the output_token to the service.

The service passes the received token as the input_token argument to Accept_sec_context(), which recognizes it as a request for TGT. (Note that current Kerberos V5 defines no intra-protocol mechanism to represent such a request.) Accept_sec_context() returns RETRY_NEEDED major_status and provides the service's TGT in its output_token. The service sends the output_token to the client.

The client passes the received token as the input_token argument to a retry of Init_sec_context(). Init_sec_context() caches the received service TGT and uses it as part of a service ticket request to the Kerberos authentication server, storing the returned service ticket and session key in conjunction with the context. Init_sec_context() builds a Kerberos-formatted authenticator, and returns it in output_token along with COMPLETE return major_status. The client sends the output_token to the service.

Service passes the received token as the input_token argument to a retry call to Accept_sec_context(). Accept_sec_context() verifies the authenticator, provides the service with the client's authenticated name, and returns major_status COMPLETE.

Sign(), Verify(), Seal(), and Unseal() as above.

3.2.3 X.509 Authentication Framework

This example illustrates use of the GSSAPI in conjunction with public-key mechanisms, consistent with the X.509 Directory Authentication Framework.

The Acquire_cred() call establishes a credentials structure, making the client's private key accessible for use on behalf of the client.

The client calls Init_sec_context(), which interrogates the Directory to acquire (and validate) a chain of public-key certificates, thereby collecting the public key of the service. The certificate validation operation determines that suitable signatures were applied by trusted authorities and that those certificates have not expired. Init_sec_context() generates a secret key for use in per-message protection operations on the context, and enciphers that secret key under the service's public key.

The enciphered secret key, along with an authenticator quantity signed with the client's private key, is included in the output_token from Init_sec_context(). The output_token also carries a certification path, consisting of a certificate chain leading from the service to the client; a variant approach would defer this path resolution to be performed by the service instead of being asserted by the client. The client application sends the output_token to the service.

[5] This scenario illustrates a different use for the RETRY_NEEDED status return facility than that described in Section 3.1 for purposes of mutual authentication; note that both uses can coexist as successive operations within a single context establishment operation.

The service passes the received token as the input_token argument to Accept_sec_context(). Accept_sec_context() validates the certification path, and as a result determines a certified binding between the client's distinguished name and the client's public key. Given that public key, Accept_sec_context() can process the input_token's authenticator quantity and verify that the client's private key was used to sign the input_token. At this point, the client is authenticated to the service. The service uses its private key to decipher the enciphered secret key provided to it for per-message protection operations on the context.

The client calls Sign() or Seal() on a data message, which causes per-message authentication, integrity, and (optional) confidentiality facilities to be applied to that message. The service uses the context's shared secret key to perform corresponding Verify() and Unseal() calls.

4 Recommendations and Future Work Areas

In order to utilize the GSSAPI atop existing, emerging, and future security mechanisms, a mech_type registration procedure must be established. Concrete data element formats must be defined for those mech_types, preferably including a common format type descriptor to distinguish the mech_type for which a token is appropriate.

Calling applications must implement formatting conventions which will enable them to distinguish GSS-API tokens from other data carried in their application protocols.

In support of application portability across general heterogeneous distributed computing environments, we recommend that an appropriate forum select or define an appropriate common hierarchic naming format, and that such a format either be supportable directly by the mechanisms underlying the GSSAPI or that suitable translation mechanisms be agreed upon.

5 Acknowledgments

This proposal is the result of a collaborative effort. Acknowledgments for contributions to this version are due to Kannan Alagappan, Doug Barlow, Bill Brown, Cliff Kahn, Charlie Kaufman, Butler Lampson, Richard Pitkin, and Joe Tardo of Digital Equipment Corporation. John Kohl, Jon Rochlis, and Jeff Schiller of Project Athena and MIT contributed valuable insights. Joe Pato and Bill Sommerfeld of HP/Apollo, Walt Tuvell of OSF, and Bill Griffith and Mike Merritt of AT&T, provided inputs which helped to focus and clarify API directions. Precursor work by Richard Pitkin, presented at meetings of the Trusted Systems Interoperability Group (TSIG), helped to demonstrate the value of a generic, mechanism-independent security service API.

APPENDIX A

PACS AND AUTHORIZATION SERVICES

Consideration has been given to modifying the GSSAPI service interface to recognize and manipulate Privilege Attribute Certificates (PACs) as in ECMA TR/46, carrying authorization data as a side effect of establishing a security context, but no such modifications have been incorporated at this time. This appendix provides rationale for this decision and discusses compatibility alternatives between PACs and the GSSAPI which do not require that PACs be made visible to GSSAPI callers.

Existing candidate mechanism types such as Kerberos and X.509 do not incorporate PAC manipulation features, and exclusion of such mechanisms from the set of candidates equipped to fully support the GSSAPI seems inappropriate. Inclusion (and GSSAPI visibility) of a feature supported by only a limited number of mechanisms could encourage the development of allegedly portable applications which would in fact have only limited portability.

The status quo, in which PACs are not visible across the GSS-API interface, does not preclude carrying PACs transparently, within the tokens defined and used for certain mech_types, and storing those PACs within peers' credentials and context-level data structures. While invisible to API callers, such PACs could be used by operating system or other local functions as inputs in the course of mediating access requests made by callers. This course of action allows dynamic selection of PAC contents, if such selection is administratively-directed rather than caller-directed.

In a distributed computing environment, authentication must span different systems; the need for such authentication provides motivation for GSSAPI definition and usage. Heterogeneous systems in a network can intercommunicate, with globally authenticated names comprising the common bond between locally defined access control policies. Access control policies to which authentication provides inputs are often local, or specific to particular operating systems or environments. If the GSSAPI made particular authorization models visible across its service interface, its scope of application would become less general. The current GSSAPI paradigm is consistent with the precedent set by Kerberos, neither defining the interpretation of authorization-related data nor enforcing access controls based on such data.

The GSSAPI is a general interface, whose callers may reside inside or outside any defined TCB or NTCB boundaries. Given this characteristic, it appears more realistic to provide facilities which provide "value-added" security services to its callers than to offer facilities which enforce restrictions on those callers. Authorization decisions must often be mediated below the GSSAPI level in a local manner against (or in spite of) applications, and cannot be selectively invoked or omitted at those applications' discretion. Given that the GSSAPI's placement prevents it from providing a comprehensive solution to the authorization issue, the value of a partial contribution specific to particular authorization models is debatable.

An Expert Systems Approach to Security Inspection of UNIX

Joseph Kuras, DEC

XSAFE is an expert system for proactive security inspection of remote Digital operating systems, which include Ultrix & VMS. Using the network for remote access, XSAFE inspects the soundness of the protection mechanism of a remote system by launching a controlled intrusion against that system.

XSAFE detects security weaknesses of a Digital operating system in six areas:

- Password Inspection,
- System File Protection Inspection,
- Account Inspection,
- Security Patch Inspection,
- Software Tool Inspection,
- User Application Inspection.

XSAFE can be used to establish a centralized network and system security audit system, as security inspection reports are generated based upon the findings of XSAFE. The inspection results provide valuable information to network and system management about further security improvements to their systems.

A Survey of Secure Unix Operating Systems

Raymond M. Wong

Oracle Corporation

1. Introduction

Since the publication of a number of early papers describing the design of secure operating system Unix prototypes and a Trusted Computer System Evaluation Criteria (TCSEC) B2 feasibility study for Unix the industry has seen the introduction of a number of secure Unix products and the evaluation of one of these products by the National Computer Security Center (NCSC) at the TCSEC B1 class. In addition, a number of standards bodies including IEEE POSIX 1003.6, TRUSIX, Open Software Foundation, Unix International, X/Open, and 88Open have all taken up the issue of standardization of secure Unix features. We can expect that in 1990 and certainly by 1991 almost every major Unix operating system vendor will have introduced either a C2 or B1 class compliant product. Furthermore, developments in secure Unix technology are likely to continue as evidenced by AT&T's intent to introduce a B2 class compliant secure Unix with System V release 4.0 Enhanced Security Package sometime in 1991 and OSF's plans to meet B3 in their OSF/2 operating system.

As an Independent Software Vendor (ISV), Oracle is committed in providing RDBMS products that are available on a broad base of vendor platforms and operating systems. With the widespread availability of secure Unix platforms in the near future, Oracle has committed itself to the development of a secure RDBMS product named Trusted Oracle. This paper is a summary of a recent study conducted by Oracle in determining the similarities and differences in the various secure Unix implementations that are now or will shortly be available. The results of this study have been used to structure the software architecture of Trusted Oracle. A major design goal of the software architecture is to ensure that a majority of the trusted RDBMS software is system independent and only a small fraction of the software needs to be modified depending on the particular operating system.

The five systems that were surveyed for this study are:

1. AT&T System V/MLS, Release 1.1.
2. SunOS MLS[tm], Version 1.0.
3. SecureWare Security Module Package Plus, Version 1.0.
4. Addamax B1st Trusted System Kit for System V 3.0, Release 1.0.
5. Trusted Xenix, Version 1.0.

All the products except Trusted Xenix are targeted at the TCSEC B1 class. Trusted Xenix is being evaluated for conformance at the B2 level. AT&T V/MLS, SecureWare, and Addamax are each being licensed by a half dozen to a dozen other vendors of Unix platforms.

2. Conclusions

The five systems surveyed in this study are similar in that they share common design goals:

- security features - providing B1 and higher level features.
- compatibility - application compatibility of the secure version with generic versions of the vendor's operating system.
- transparency - single level users should be minimally impacted by the security features.
- standards - adherence to standards when they exists.

Although the systems share the above goals, we have seen that they differ greatly in how they implement each of the security features. Ranging from how permission bits and ACLs interact, to how IPC objects are treated for mandatory access control, to how audit events are selected, there is great diversity in how each of these features is implemented on each of the five systems.

Independent software vendors like Oracle have been accustom to differences in Unix from vendor to vendor. However, as Unix migrates to standards such as System V release 4.0 or OSF-1, and as application interface standards are adopted these differences become less drastic and fewer in number. Except for the standards work in IEEE POSIX 1003.6 which is still on going, there is no universally accepted model for how secure Unix should handle each of the security requirements required by the TCSEC. No standard application interface definition exists yet for secure Unix. Consequently, ISVs will discover that secure Unix systems are fairly compatible with generic versions of the vendor's operating system. Most untrusted applications can be made to run on secure Unixes with some modifications or in some cases with no modifications. However, secure Unixes will differ greatly from one another. Changes made on one secure Unix to make an application work will not necessary succeed on another system. Portable trusted applications for secure Unix are difficult not only because trusted may mean something different on different systems, but also because each of the systems differs in their discretionary and mandatory access control policies and mechanisms.

The future of secure Unix operating systems will likely see work in each of the following areas:

- standardization - for example agreement on how to treat ACLs, multilevel directories, IPC objects for MAC, and event selection for auditing.

- windowing - driven by the market demand for bit-mapped graphics applications, vendors will need to provide multilevel windowing capabilities. This will also be accelerated by the demand for Compartmented Mode Workstations.

- networking - most vendor products only address standalone systems today. Since most user environments not only involved a distributed system but a heterogeneous one, standards for interconnecting secure Unixes from different vendors are a necessity.

- audit analysis - manual analysis of the volumes of information generated by auditing systems using primitive pattern matching tools is burdensome. Improvement in the tools for automated audit trail analysis and perhaps real-time audit analysis are needed.

Roles for Users and Privileges for System Processes: High-trust Mechanisms for Low-Trust Systems

David L. Gill

Position Paper

To provide more trust for systems being developed to meet the C2 Class of Trusted Computer Systems Evaluation Criteria (TCSEC), a technique is suggested for systems providing audit; identification and authentication; and discretionary access control of and secure reuse of objects.

The technique is to "borrow" concepts from the B and A division of the TCSEC for use at the C division. These include:

- roles for general user which restrict the domain of execution for each user.
- separation of operator, security officer, and system administrator roles which ensure checks and balances for security related operations. It is assumed that general use of user functions would be constrained while in each of these roles.
- the processes executed by users logged in at the security roles listed above should have fine-grained privileges associated with them. These privileges should explicitly indicate the aspect of the security policy which is to be bypassed during execution.
- to ensure access to the security kernel for execution of secure processes, the trusted path mechanism shall also be the means of assess to these security modules

The Defense Intelligence Agency (DIA) has developed a set of requirements known as the Compartmented Mode Workstation (CMW) requirements. These requirements take as a basis the Labeled Security Protection (B1) Class of the Department of Defense TCSEC and augment it with accountability and assurance requirements from the B2, B3 and even A1 classes of the TCSEC.

This presentation will discuss the TCSEC requirements used for defining the ones listed above. It will give rational for consideration of such requirements in a C2 system, and discuss alternatives for implementation of the requirements listed above.

Beyond Bell-LaPadula: A Security Model for Real Applications

Pat Bahn

ABSTRACT

The Bell-LaPadula model was proposed in the early 1970's. At this point it is obvious that no application of merit can conform to this model. The model assumes that all objects are a homogeneous security level. The Bell-LaPadula model also assumes a simplistic method of security that conforms only to certain military applications. Bell-LaPadula does nothing for originator control or other special caveats, nor could Bell-LaPadula be applied to serious work such as medical or commercial records.

The author proposes that a finer level of detail be applied, that objects be characterized by internal datatypes with attached security labels and that emphasis be placed on spheres of interest along the discretionary access authority and that access be granted often on certain value ranges.

BUILDING GENERALIZED ACCESS CONTROL ON UNIX®

Marshall D. Abrams *
Leonard J. La Padula †
Ingrid M. Olson *

The MITRE Corporation
* 7525 Colshire Drive, Mc Lean, VA 22102
† Burlington Road, Bedford, MA 01730

Overview

This paper describes an approach for studying and constructing access control policies for automated information systems. It presents a view of access control policies as *rules* specified in terms of *access control information* and *context* by *authorities*. Expressed in terminology introduced in [ISO89], these concepts represent dimensions of choice and constraints to the designer of a system:

- Access Control Information (ACI) — Characteristics or properties of subjects and objects. Their names are used in specifying the rules of the system; their values are used by the rules to determine whether a subject may access a specific object.
- Access Control Context (ACC) — Additional information, such as time of day, that is used in access control decision making.
- Access Control Authorities (ACA) — agents who specify ACI, context, and rules.
- Access Control Rules (ACR) — the set of formalized criteria for adjudicating requests by subjects for access to objects.

Using these primitive elements, a large set of policies can be formulated in a manner appropriate for enforcement in automated systems.

This paper presents the Generalized Framework for Access Control (GFAC). Using this framework to examine the similarities, differences, strengths, and weaknesses of existing policies, we derived general concepts that may improve existing policy models and create new policy models, leading to improved access control mechanisms. Access control policies such as Mandatory Access Control (MAC), Discretionary Access Control (DAC)[1], and Clark-Wilson Integrity [CLAR87] are merely possible points in a broad space of access control policies. Work is underway to implement a prototype demonstrating GFAC controls using AT&T System V/MLS as a base. A comprehensive informal description of GFAC may be found in [ABRA90]. A preliminary formal description may be found in [LAPA90].

® UNIX is a registered trademark of AT&T

[1] In this paper, MAC and DAC are treated as reserved words referring to Mandatory Access Control and Discretionary Access Control respectively, as defined in [NCSC85].

Access Control Information (ACI)

ACI is associated with subjects and objects. The names of ACI are used in specifying the security rules of a system; the values of these ACI are used by the rules to determine whether a given subject may access a specific object. A set of named ACI is associated with a class of subjects or objects and a particular access control policy.

ACI related to subjects may include, but are not limited to, the following types of information:

- identification (e.g., user ID, name, employee number)
- authentication information (e.g. password, smart card PIN, fingerprint)
- biographic information (e.g., department, nationality)
- clearance
- location
- access permissions relative to classes of objects (e.g., capabilities)
- role (e.g , user, system administrator, security officer).

ACI related to objects may include, but are not limited to, the following types of information:

- classification
- handling restrictions (e.g., PERSONAL FOR)
- classification authority
- source/originator
- document number
- owner
- programs allowed to access the object and their access permissions (e.g., Clark-Wilson model to enforce the well-formed transaction)
- identities of users and their access permissions (e.g., access control list).

Access Control Context (ACC)

The ACC contains information that is not associated with a subject or object, but is necessary to the access control decision process. The information becomes security relevant by virtue of being included in the rules. The integrity of this context information must be protected by preventing unauthorized changes. Security policy may also require secrecy protection.

Access control context may include, but is not limited to, the following types of information:

- Time — access to the information (i.e., sensitivity of the information) may vary with time (e.g., the Department of Labor Statistics information on last month's unemployment rate is sensitive until 9:00 am on Tuesday morning when it is made public)

- Status — the access control restrictions depend on a status variable which is officially changed to reflect some condition in the real world (e.g., crisis or exercise status)
- Group membership — the names of groups are ACI associated with targets, but the definition or enumeration of membership in a group is part of the context.

Access Control Rules

In a secure AIS, access to an object by a subject is controlled by a Trusted Computing Base (TCB) [NCSC85]. The TCB determines which subjects may access which objects with what kind of access permissions. Such a system will often employ some set of system functions (e.g., open file, activate process, delete file) as part of the TCB. As a part of the normal operation of these functions, they also adjudicate the implicit request for access by the subject to the object.

These system functions are sometimes referred to as the security or access control "rules" of the system; however, this terminological convention tends to be confusing.[2] In this paper, we use the term *rule* to identify only the portion of the function that adjudicates the access control requests. That is, we separate the system function into two operations: one in which adjudication of the access is requested (i.e., the rule is invoked) from some TCB-resident security "rule-base," and a second that performs the requested non-policy-related functions (e.g., establishing access between a subject and a file, initializing a new subject, or removing a file object from the system). In general, access control rules are of the general form:

> A subject is permitted to access an object in access mode M only if the ACI of the subject is compatible with the ACI of the object.

An important concept in creating new subjects and objects is "inheritance" [NCSC89]. A new object may be simply a copy of an old object, may be created anew, may be created by changing or editing an existing object, or may be formed by combining two or more existing objects. Inheritance rules are concerned with establishing the ACI associated with the new object. The MAC inheritance rule may be inferred from the TCSEC in a rather straightforward manner: a new object is labeled at the sensitivity level at which the user is operating, usually the level at which he or she logged in.

There does not appear to be any DAC inheritance rule in the TCSEC. This appears to be a consequence of the discretionary aspect. Some implementations provide defaults. UNIX for example, allows the user to specify default user/group/world (UGW) protection for all new objects. A new object, created by editing an old object belonging to the user, usually retains (inherits) the old object's ACI. But when the new object is created by a different user, the original user's ACI are no longer associated with the object.

[2] The "rules" described in the Multics interpretation of the Bell-LaPadula model [BELL75] can be seen to include some of the non-policy-related functionality described above.

Authority

One may associate the notion of span of authority with originators, owners, Information System Security Officer (ISSOs), commands/agencies, and national or corporate policy. In general, higher authority levels will be responsible for establishing the policy, information system architects will translate the policy into rules, the system modelers/designers will represent these rules in a formal manner and will design their implementation, and the ISSO will be responsible for entering and maintaining the subject/object ACI. The structure of the authority for a given system will be determined in part by the rules established for that system. For example, consider whether the ISSO is allowed to delegate some of his authority to owners and the exact form and extent of that delegation.

Control of access to the ACI is essential. Control of the ability to read and modify ACI is the key to the strength of the access control mechanism. It is convenient to think of ACI as objects with an access control tree attached to selected ACI (objects). It may not be necessary to build an access control tree for every attribute. These access control trees define the authority and privileges in the system.

It appears that providing three levels of hierarchy is reasonable, although one could imagine more or fewer levels depending on the particular policy or on some practical limitations. In general, the ISSO may be viewed as the ultimate authority within the AIS. In some cases, the ISSO may delegate authority to others.

Prototyping Plans

One of the decisive factors that led us to select System V/MLS as the base on which to prototype GFAC was the modular implementation that AT&T employed to add multilevel security to System V. The interpretation of (and operations performed on) labels were isolated in a kernel-level module known as the *MLS module* [FLIN88]. The MLS module was implemented as a removeable and replaceable portion of System V/MLS. We plan to take advantage of this modularity by modifying or replacing the MLS module to implement selected ACI and ACR, implementing "interesting" policies.

Many "interesting" policies are discussed in [WILL88]; commercial counterparts are easily identified. Many of these policies can be implemented in B1 systems. While this is expedient, it does not meet our research objectives. However, we certainly applaud any implementation of additional access control policies.

Summary

This paper is a snapshot of our thinking about GFAC, identifying major components — Access Control Information (ACI), Access Control Context (ACC), Access Control Rules (ACR), and Access Control Authority (ACA) — in the design space of access controls. By making design decisions about each of these variables and their combinations, alternative access control policies can be implemented. GFAC provides an improved framework for expressing and integrating multiple policy components.

Bibliography

ABRA90

Abrams, M.D., K. W. Eggers, L. J. LaPadula and I. M. Olson, "A Generalized Framework for Access Control: An Informal Description," *Proceedings of the 13th National Computer Security Conference,* Baltimore, MD, October 1990.

BELL75

Bell, D.E. and L.J. LaPadula, *Secure Computer Systems: Generalized Framework for Exposition and Multics Interpretation,* ESD-TR-75-306, The MITRE Corporation, July 1975. (Also available though National Technical Information Service, Springfield, VA, NTIS AD-A023588.)

CLAR87

Clark, D.D. and D.R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," *Proceedings of the 1987 Symposium on Security and Privacy,* Oakland, CA, IEEE Computer Society Press, April 1987, pp. 184-194.

FLIN88

Flink, C. W. and J. D. Weiss, "System V/MLS Labeling and Mandatory Policy Alternatives," *AT&T Technical Journal,* May/June 1988, pp. 53-64.

ISO89

International Standards Organization, International Electrotechnical Committee, Joint Technical Committee 1, Subcommittee 21, *Working Draft on Access Control Framework,* document number 4206, December 1989.

LAPA90

LaPadula, L.J., "Formal Modeling in a Generalized Framework for Access Control," *Proceedings of the Computer Security Foundation Workshop III,* 12 June 1990.

NCSC85

National Computer Security Center, *Department of Defense Trusted Computer System Evaluation Criteria,* DOD 5200.28-STD, December 1985.

NCSC89

National Computer Security Center, *Final Evaluation Report of American Telephone and Telegraph System V/MLS Release 1.1.2 Running on UNIX System V Release 3.1.1,* CSC-EPL-89/003, 18 October 1989.

WILL88

Williams, J.C. and M.L. Day, "Sensitivity Labels and Security Profiles," *Proceedings of the 11th National Computer Security Conference,* 17-20 October 1988, pp. 257-266.

PACL's: An Access Control List Approach to Anti-Viral Security[†]

David R. Wichers[††] Douglas M. Cook Ronald A. Olsson
John Crossley Paul Kerchen Karl N. Levitt Raymond Lo

Division of Computer Science
Department of Electrical Engineering and Computer Science
University of California, Davis
Davis, CA 95616

(916) 752-7004

Abstract—Almost all attempts at anti-viral software have been a reaction to specific viruses that have infected the user community. These solutions attempt to protect against a specific strain or strains of viruses rather than provide general protection against a wide variety of viruses. This paper describes a new, conceptually simple approach that provides a more general solution to the virus problem. Our approach associates with each file in a system an access control list (ACL) that explicitly specifies which programs can modify the file. Thus, a virus cannot modify arbitrary files and its possible effects are greatly reduced. Our approach is unique in the way it uses ACL's to specify which *programs* can access a file; other schemes use ACL's to specify which *users* can access a file and how. We use the acronym *PACL*'s, for Program ACL's, to refer to these ACL's and to our scheme. To see how our ideas can be incorporated into an existing operating system, we have designed an extension to the UNIX[†††] kernel. We also constructed a simulator that has allowed us to gain operational experience with our ideas in a typical user environment. The results indicate that our scheme is a promising approach for preventing the spread of viruses without being too intrusive on users.

[†] This research supported by Lawrence Livermore National Laboratory, the State of California MICRO program, and Deloitte, Haskins, and Sells, Inc.

[††] David Wichers' present address is: Arca Systems, Inc., 2841 Junction Ave, Suite 201 San Jose, CA 95134 (408) 434-6633

[†††] UNIX is a registered trademark of AT&T.

1. Introduction

A computer virus is a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself [6]. One attribute of viruses that allows them to spread so easily is that a virus inherits all of a user's privileges when the user runs an infected program. Typical operating system protection schemes provide no help in such a case—they protect a user's files from other users, but not from him/herself. Thus, a virus can quickly infect all of a user's files. Even worse, if the user has special system privileges (e.g., 'superuser'), the virus can infect all files on a given system.

A typical virus propagates itself by searching for an uninfected program and copying the viral part of its code into that program so that when the newly infected program is run, the viral code will be executed. To prevent propagation, viruses must be prevented from inserting themselves into other programs. (We assume that the operating system prevents programs, including viruses, from writing directly to disk.)

Two simple observations form the basis of our approach. First, the typical virus carrier is unrelated to the programs that it infects. Second, programs executing on behalf of a user have more privileges than are necessary to complete their assigned task. For example, an infected game program might have the privilege to access all of a user's files. Yet it should only have access to those related to the game, e.g., a score file. Our approach, then, is to restrict a program's privileges to the minimum needed to complete its assigned task. Then, if a program is infected, it will not be able to infect unrelated programs (files).

To impose this *least privilege* restriction, we associate an access control list (ACL) [7, 9] with each file in the system. In our scheme, a file's ACL contains the names of all the programs that may modify the file. We use the acronym *PACL*'s, for Program ACL's, to refer to these ACL's and to our scheme. Thus, to modify (write, append, delete, etc.) the file, a program must be on the file's PACL. Our use of PACL's differs from that found in standard ACL schemes: we store names of *programs*, as opposed to the names of *users*, that can access each file.

The notion of least privilege fits well with common system usage. Users create files using a number of different programs. These files are usually modified only by the programs that create them. For example, consider the typical steps involved in creating, compiling, and linking a C program. To create the program, the user uses his/her favorite editor to create source files. During the entire life of those files, they are only modified by the same editor that created them. When these files are compiled, the compiler generates object files. Each time the program is recompiled, these object files are written over by the same compiler, and not by any other program. Similarly, the linker creates the executable and writes over the executable file each time the program is relinked. This usage suggests that normal files are modified by a small number of programs, usually only one. Of course, more complicated usages exist, but they are less common.

Since the number of programs that need to modify a single file is usually very small, we can keep track of these programs in order to prevent other programs from deliberately or accidentally modifying files. This method is similar to existing computer protection mechanisms based on access control lists. The standard ACL scheme is designed to control how each user's files can be accessed by other users. That is, a file's ACL indicates what users may access the files, and in what ways. If the ACL does not explicitly state that a user is allowed to perform the function requested, then it is not allowed. The difference between this security problem and the virus problem is that a virus security system needs to protect a user from him/herself, not from other users. The virus problem is inherently a problem of integrity, not security. Our *PACL-Integrity* scheme is therefore simpler, associating with each file a list of all programs that can modify the file.

To see how our ideas can be incorporated into an existing operating system, we have designed an extension to the UNIX kernel that incorporates our PACL scheme. We have also constructed a simulator to allow us to gain experience with the PACL-Integrity model without requiring actual changes to the kernel. The experience we have gained shows that the scheme seems reasonable to implement and is not too intrusive on the user.

The remainder of this paper is organized as follows. Section 2 discusses our PACL-Integrity model in more depth. Section 3 describes how the model can be realized in the UNIX kernel. Section 4 presents the simulator and section 5 describes our experience using it. Section 6 discusses the tradeoffs involved in our approach and outlines future work. Section 7 summarizes related work. Finally, section 8 contains some concluding remarks.

2. The PACL-Integrity Model

The PACL-Integrity model associates a PACL with each file on the system. The PACL for a given file names all programs that have the privilege to modify the file. When a file is created, its PACL is set to contain the name of the program that created the file. During the life of the file, the file's PACL can be changed only by a trusted utility program. This utility allows a user to tailor the protection mechanism to meet his/her needs.

The success of a protection scheme depends on how intrusive users find it. A scheme that is too intrusive will effectively render a system unusable. For example, requiring an explicit acknowledgement from a user each time any file is to be accessed might be a secure scheme, but it is not usable. Moreover, if a scheme that is too intrusive provides a means by which the user can disable it, then users will simply run with security checks disabled, effectively rendering a system insecure.

To make our approach secure yet usable, we include a number of 'user-friendly' features. These features simplify common usages of the PACL-Integrity mechanism. The first feature is an inheritance mechanism that allows a user to define a default PACL for a directory. Any file (or subdirectory) created in this directory inherits the directory's default PACL, as well as the name of the program that created the file. This feature allows the user to tailor a directory to the type of work being done in it. An entire system (or account) can be tailored in this manner by creating a default at the root (or home) directory and then building directories below it.

The second feature allows a user to specify a global inheritance policy. The user can define a default PACL for any file based on its extension (suffix). For example, a UNIX object file is typically created by an assembler or compiler and given the extension '.o'. Later, the linker reads in a number of object files, links them together, and generates executable code. When it has successfully generated an executable, it sometimes will remove the object files as they are no longer needed. Since the object files were created by the compiler, their PACL's will contain the name of the compiler, but not that of 'ld' (the linker). With the extension-based default mechanism, the user can define a default for '.o' files that contains 'ld', thereby allowing the linker to remove unwanted object files after it has created the executable.

The third feature allows the user to enable/disable the PACL mechanism for a particular file. This feature is provided by associating a flag with each file. If this flag is enabled, the normal PACL security rules will be applied to that file. If the flag is disabled, then all PACL security rules for the file are ignored and only the 'normal' security rules will be used when the file is accessed; i.e., any program with appropriate access rights can modify the file.

The final feature allows a user to temporarily disable the PACL mechanism for all of his/her files. It also allows the system administrator to temporarily disable the PACL mechanism for the entire system. This feature is needed to facilitate programs that need to modify many or all of a user's or system's files. For example, a utility program that restores files from backup tapes will typically modify many files during its execution.

These features are provided to allow the system to be tailored to meet each individual user's needs. Once defaults have been set up correctly, each user should be able to use the system while being protected from viruses, without being unduly inconvenienced by the PACL mechanism.

The PACL-Integrity mechanism makes several basic assumptions about the underlying hardware and operating system. The devices on which programs are stored (e.g., disk) must be protected so that they can only be accessed by kernel code. Without such protection, a virus could write directly to a device, bypassing all protection mechanisms. This requirement rules out the possibility that this type of system would be viable in some personal computer environments where direct disk access is not protected, for example. The operating system itself must check all file accesses to make sure the PACL security rules are enforced. It must also protect the PACL's themselves from illegal modification. The hardware and operating system must also protect against standard attacks, such as modifying system buffers or kernel code.

3. A PACL-Integrity Model for UNIX

3.1. Overview

Our PACL-Integrity model can be implemented for UNIX by extending the kernel. The PACL scheme must be included in the kernel to ensure that all file accesses are checked. The current UNIX protection mechanisms, based on user names, are still enforced. If an attempt to write satisfies the existing security rules, the PACL mechanism then further verifies the validity of the access.

When a file is created, its PACL is created as well. A file's PACL is stored as part of the header information (i.e., *inode*) of the file, just like the mode bits, owner, size, date, and time fields. Since the PACL is part of a file's inode, the PACL information for a file is removed when the file is deleted, which simplifies the task of PACL maintenance.

The kernel builds the PACL for a new file from three items. The first item put in the PACL is the name of program that creates the file. In UNIX, a program's name is its complete pathname. For example, the editor program 'vi' in the directory '/usr/ucb' has the name '/usr/ucb/vi'. The second item put in the PACL is the default PACL of the directory in which the file is created. The final item put in the PACL is the default, if any, for the new file's extension. (Note that the defaults put in the PACL are those in effect when the file is created; if the defaults are later changed, the PACL's of existing files are not modified automatically.)

The specific kinds of access for which the kernel must check include opening a file for writing and unlinking a file. The former gives the program the privilege to modify the file in any manner while the latter deletes the file. We consider deletion a form of modification.

3.2. New System Calls

Nine new system calls give programs the ability to interact with the PACL mechanism. The first system call, *setppriv()*, is a privileged call that sets the state of the current process into a mode that allows it to call the other new system calls. (This method is analogous to a process setting its user-id to root in regular UNIX.) Without executing this initial call, a process is not allowed to use any of the other system calls that interact with the PACL's, with one exception described below; in such a case, they simply return an error to the calling process. The only programs that are allowed to use *setppriv()* are the programs listed in the file '/etc/paclprivs'. One example of an entry in this file is the utility program described later.

The second call, *paclenable()*, is used to enable or disable (based on its argument) the entire PACL mechanism for the given process and its children. If the initial system process (*init*) disables the PACL mechanism, then the effect is that the PACL mechanism is disabled for the entire system since all processes are children of *init*.

The third call, *clrppriv()*, removes a process from PACL privileged mode. It allows the process to relinquish its privilege when no longer needed. The two calls *setppriv()* and *clrppriv()* allow programs to create critical regions in their code where they have privilege to access PACL's. Outside of these regions, PACL privileges are not necessary and hence should not be enabled.

The fourth call, *getppriv()*, is the only call that will not return an error if *setppriv()* has not been previously called. It tells the currently running process whether or not it is currently in PACL privileged mode, i.e., the process successfully called *setppriv()* without calling a corresponding *clrppriv()*.

The next two new system calls allow a program to manipulate PACL's. Only the owner of a file can change its PACL. The first, *addpacl()*, adds a program name to a given file's PACL. The second, *delpacl()*, deletes a program name from a given file's PACL. These system calls also allow a file's owner to enable/disable the PACL mechanism for a particular file.

The remaining three system calls allow a program to query a file's PACL in various ways. These calls can only be executed by the file's owner. The first, *getpacl()*, returns a list of all the program names in a file's PACL. The second, *verpaclm()*, determines if a specified program has the privilege to modify a given file. It compares the program name with those in the file's PACL, handling links if the filename provided is a link to another file. The third, *verpaclr()*, determines if the specified program has the privilege to remove a given file. It is similar to *verpaclm()* except it does not traverse links because any remove reference to a link would be removing the link, and not the file to which the link points.

3.3. The *ch* Utility

The above eight system calls provide the means for a system program to manage PACL's. The utility program, *ch*, described below uses these calls and is an example of a type of user interface that can be provided for user interaction with this mechanism. *ch* is listed in '/etc/paclprivs' so that it is authorized to use these PACL system calls on the user's behalf.

To use the *ch* utility, the user must first enter his/her password. We make the assumption that a virus can assume a user's login name but it does not know the user's password. Otherwise, we cannot distinguish a virus from a legitimate user.

ch allows the user to:

- add/remove program names from PACL's;
- display the contents of PACL's;
- set/clear the enable flag in PACL's;
- modify the default PACL's for directories and file extensions; and
- temporarily turn off the entire PACL mechanism (e.g., for that user during a single login session).

These features correspond to those described in section 2. Several additional features make the utility more usable. One feature allows the user to traverse the directory structure; a user can, therefore, move to different directories without exiting the utility. A second feature is that *ch* provides all the remove privileges that exist in a normal shell. The 'rm' (remove) program may not have privilege to remove most files; i.e., it may not be in the PACL for every file. The utility, therefore, provides an 'rm' command with functionality equivalent to that of the 'rm' program. Without such a command, the user would need to add the 'rm' program to a file's PACL, exit the utility, and then use the 'rm' program to remove the program. For the same reason, the utility also provides an 'rmdir' (remove directory) command. Basically, *ch* provides a subset of the normal shell commands along with the features described above that allow the user to tailor the PACL security system. If the user executes a program from within *ch*, a new process is created to execute that program. This process is subject to the rules that apply to the new program, not those that apply to the *ch* program. Other utility programs can easily be generated by the system administrator by writing programs using these system calls and then adding the program names to '/etc/paclprivs'.

3.4. The Role of the Superuser

In existing UNIX systems, the superuser—e.g., the 'root' account—may bypass the normal protection mechanisms. Having root privilege is not sufficient to override the PACL protection mechanism in our system. In particular, a user (or would-be virus) executing as root can only disable the PACL mechanism using the *ch* utility, for which it must give the root password. A program running as root must, therefore, be listed in a file's PACL in order for that program to have the privilege to modify that particular file. This approach limits the damage potential of a virus that somehow acquires root privilege.

This restriction, however, requires us to change the current method by which the superuser changes the root password. Currently, the superuser uses the 'passwd' program to change the root password. The password program prompts for the new password without asking for the old one. Thus, any user (or program) that acquires root privilege can change the root password without knowing the previous password. Such a user could then use *ch* to break system security. Therefore, we now require *passwd* to ask the superuser for the old root password before changing it. This additional requirement prevents a virus from changing the root password without knowing the previous password. The one exception is that the superuser can change the root password without entering the old password when the system is brought up in console (single user) mode. This exception exists to allow access to a system in case its password file gets corrupted.

Since the success of our anti-viral scheme depends heavily upon password security, viruses must be prevented from obtaining passwords. In our scheme, the password file itself is protected so the only programs allowed to modify it are 'passwd' and those that modify information about users (e.g., user names, phone numbers, etc.). A new user can be added according to one of two methods. The first method is to add an editor, say 'vi', to the password file's PACL, then edit the file to include the new user, and then remove 'vi' from the PACL. This method is not a major inconvenience to the system administrator if new users are added infrequently. On the other hand, the above method is cumbersome for a system where new users are added frequently. A better method, then, is to write a new utility program that is specifically designed to add users to the password file and to place the name of this new utility in the password file's PACL. Execution of this utility program should be restricted to only the system administrator.

4. A PACL-Integrity Model Simulator

We constructed a UNIX-based simulator to allow us to experiment with our ideas. Building a simulator required less effort than making kernel modifications would have. Doing so also had no impact on other users of the system as making kernel modifications would have.

The simulator consists of modifications to the standard C library. It is not a program itself. The simulator library contains modified versions of the normal system calls that deal with files (e.g., open) and code for the new system calls dealing specifically with PACL's. The normal system calls take the same arguments as usual. Thus, the simulation environment is transparent to most programs; they just need to be linked with the new library. The code in the library routine that handles a normal system call is an interface to the original routine that handles the system call. It first does whatever PACL checking is needed and then calls the original routine, which has been renamed.

The simulator maintains a *virtual root*. The virtual root allows any directory to act as the root directory during simulation experiments. The simulator maps any reference to root (i.e., a pathname that starts with '/') to the virtual root. For example, if '~cook/test/pacl' is the virtual root, the simulator maps '/bin/cp', the copy program, to '~cook/test/pacl/bin/cp'. Using a virtual root lets us test the PACL mechanism by defining a subdirectory that contains an entire UNIX environment, i.e., all the standard system programs. The programs in such a subdirectory are linked with the simulation library. Using a virtual root also allows a user to experiment on a system without requiring root privileges. Further, it allows several users to run experiments at the same time as each one can define their own virtual root. (The idea of a virtual root is similar to the UNIX 'chroot' command except it works on a per-process basis and does not require root permission.)

Section 3 described how a file's PACL information is stored as part of its inode. That is not possible without kernel modifications. The simulator, therefore, stores the PACL information for file *x* in another file, *x.pacl*. Similarly, the default directory PACL for a directory is stored in the file 'default.pacl'. These files are not visible to the user when running under the simulator. They can only be created by the simulator for its purposes.

The default PACL information for file extensions is stored in an environment variable. The simulator uses this information along with the default directory PACL information and the executing program's name (see below) when creating the PACL for a new file.

The simulator needs the name of the currently executing program for creating PACL's and comparing access rights. The simulator maintains that name in an environment variable. The variable is set in the simulator library's *execve()* system call, which is invoked whenever a process is created. It is examined whenever a process executes a system call that needs PACL privilege. This method is insecure because a process can modify its environment variables—e.g., a process can change the simulator's idea of its name to gain illegal access to a file. However, the method is adequate for our simulation purposes. In a kernel implementation, the name of the currently executing process would be stored so that only the kernel could modify it.

5. Experience

The additions to the C library to form the simulator library required about 1000 lines of code. The additional code intercepted system calls, translated pathnames to virtual root-based pathnames, and checked PACL permissions.

We have used the simulator in a number of situations and have gained some idea as to the effectiveness and intrusiveness of our PACL scheme. Our tests fall into two categories: general interactive use and installation of software systems. In the first kind of tests, users performed the activities they normally would on a system—i.e., developing programs, writing papers, etc.—and would also occasionally attempt to defeat the PACL mechanism. In these tests, the PACL mechanism worked as it was intended: It was successful in preventing simulated viral attacks without being too intrusive. One observed drawback, however, was that users needed to be aware of the PACL mechanism. One common problem, for example, was that users had problems removing files since the remove program 'rm' was not on the PACL of the file being removed. The utility program proved useful, but it requires the user to learn a new tool.

The second kind of simulator test—software installation—was also generally successful. We attempted to install two large software systems, GNU Emacs [11] and the SR concurrent programming language [1], in the simulated environment. Although the installations uncovered several problems with our simulator, they did demonstrate the validity of the overall design of our PACL scheme.

6. Discussion

Our PACL-Integrity model is an integrity model only. As such, it protects files from illegal modification but not from exposure. One advantage of it being just an integrity model is that the system is greatly simplified. PACL checks occur when the file is opened for writing and the checks themselves are very fast. A PACL check consists of looking up the program name to see if it exists in the file's PACL. Since a file's PACL will typically be very short, that check will be very fast and the space overhead involved per file will be minimal. (For simplicity, we have restricted in our initial designs a fixed sized space to store the PACL for each file.)

Our PACL scheme is obviously not perfect. It can be defeated by exploiting existing operating system security loopholes or trojan horses. It also requires that the PACL's for the system are set up correctly, which requires user and system administrator cooperation.

One potential vulnerability of our PACL scheme is that a virus could invoke other, more trusted programs to do its dirty work. For example, a virus could send commands to infect files to 'vi'. One possible solution to this problem would be for programs to impose restrictions on how they operate; e.g., 'vi' might accept only interactive input rather than accepting input from another program. A more general solution, however, will require further study. Even with this vulnerability, our PACL scheme substantially reduces the vulnerability of the overall system.

One issue that we have not fully resolved is exactly what constitutes the name of a program. As described earlier, the name of the file is its complete pathname. However, a single file on disk can have many different names (i.e., paths to it) through *hard* or *symbolic* links. A hard link establishes another name for a file by having another directory entry point to the file's inode. A symbolic link is a file whose contents is a file name; when the symbolic link is opened, the kernel instead opens the contents of the link. Given such possibilities of multiple names for a given file, which name or names should be used in the PACL checks needs further study.

Another issue related to naming is what to do when the name of a program changes. Since the PACL's store full pathnames of a file, they must be changed. One possible solution is to extend the *ch* utility to provide a rename option. However, that is expensive as it needs to search all PACL's for all files in the system. Moreover, it requires user intervention. Another possible solution is to store in the PACL the program's inode number instead of its name. A sequence number would also need to be maintained for each inode to distinguish between different uses of an inode, e.g., to ensure that when a free inode is reused, it does not accidentally allow access to the wrong files. Renaming is important because it occurs fairly frequently, although more often for user programs than for system programs. Another related issue is what to do when a new version of a program is installed. If program names are stored, then the PACL scheme works fine. If inode numbers are stored, then they would need to be updated, which is expensive as described above. One final related issue is how to handle deletion of programs. When a program is deleted, it should be removed from all PACL's in which it appears. Otherwise, a virus might install a new program in that place. On the other hand, if a program is deleted just before a new version of it is installed, then the cost of cleaning up all PACL's should be avoided. These issues are important and related to one another. Further work and experience is needed before the 'right' solution can be determined.

One possible objection to the entire PACL approach is that it requires future knowledge to be totally effective: it must know *for all time* what programs will need to access what files. That is clearly impossible, especially since new programs can be added to a system. For example, suppose an existing file system has its PACL lists set up so that the C compiler, say located in '/bin/cc', is allowed to create '.o' files. If an alternate C compiler such as GNU's, say located in '/usr/local/gcc', is added to the system, then the PACL's for all '.o' files should be updated to also allow the new compiler to modify those files. Requiring users to perform such modifications of PACL's is not attractive; a tool to automate such modifications should be straightforward to develop.

The use of the extension-based defaults and the directory inheritance in our PACL scheme provides a flexible enough environment for a user to perform most tasks without considering the PACL's. A more complicated task, especially one involving files with nonstandard extensions, may require the user to modify PACL defaults. However, once that is done, the task can be completed with little difficulty. User intervention is typically only required to set up defaults for a new task; repetitions of that task do not require further intervention.

The initial setup of a system that uses PACL's is also a nontrivial task. In particular, the system administrator must determine PACL's for each system file, and directory and extension defaults. Fortunately, such work needs to be done just once. Of course, this problem will go away if PACL systems become the standard; vendors would then ship PACL-equipped systems already set up.

7. Related Work

Recently, Eugene Bacic [2] proposed a similar scheme that was developed independently from that proposed in this paper. His mechanism also associates an additional ACL-like list with each data object to provide integrity controls by constraining the programs that can manipulate an object. His paper addresses the subject from a more theoretical slant than the application specific (UNIX) approach discussed here.

Karger [8] and Boebert and Ferguson [4] have proposed solutions similar to each other that attempt to address the Trojan Horse problem. Both solutions interpose a protected subsystem between programs and the filesystem to protect the filesystem. They are related to the mechanism we described in that they use some type of knowledge base (in our case the PACL's) to make access control decisions based on the user executing the program, the program being executed and the files being accessed. The methods they proposed to generate and use this knowledge base are quite different. The intent of our solution is to generate this knowledge base as simply and easily as possible, while making it simple to design, simple to build, simple to maintain, powerful to use and simple to understand.

In a landmark paper, Clark and Wilson [5] introduced a model of integrity that is based on control of which actions users can perform on particular data items. The model is applied to constrained data items (CDI's) and only allows access to these CDI's through Transformation Procedures (TP's). The central system enforced property required by Clark-Wilson is that the system maintain a list with entries which describe for a user-id, TP pair, which CDI's the user can access with the given TP. The system must further ensure that no CDI can be manipulated except through a TP. The PACL mechanism described in this paper can directly support the enforcement of Clark Wilson controls with the restriction that there is only a single list of programs allowed to access a given file rather than a separate list for each user or group of users. Using the normal access control mechanism, the users which can access a CDI are described by the standard permissions, and the TP's which can access the CDI are listed in its PACL.

8. Conclusions

The PACL scheme presented in this paper is a step toward providing protection against viruses. It is an attractive approach since it is relatively simple, both conceptually and to implement, and it aims to protect against all viruses, not just specific strains. The simulator allowed us to gain experience using our scheme. This experience has been quite positive and shows that our approach is feasible. Although this paper describes our scheme and experience in the UNIX environment, our PACL scheme also applies to other operating systems. We plan to gain additional experience using the simulator, and then to implement our scheme in the kernel. Our other plans include extending the PACL scheme to a networked environment and considering how a collection of systems—some using our PACL scheme and some not—will interact. At a broader level, we are also investigating ways of combining various protection schemes (e.g., ACL's, capability lists, type enforcement schemes [10], integrity labels [3, 10], and POSET model [6]) into one unified scheme. The unified scheme will allow flexibility in choosing which scheme is appropriate for a given problem.

Acknowledgements

Bill Wilson, Seth Abrahams, Doug Mansur, and John Wagner provided very useful input on our ideas.

References

[1] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Trans. on Prog. Languages and Systems*, 10(1), pp. 51-86, January 1988.

[2] E.M. Bacic. Process execution controls as a mechanism to ensure consistency, *Proceedings of the 5th Computer Security Applications Conference*, Tucson, AZ, Dec. 1989.

[3] K.J. Biba. Integrity considerations for secure computer systems. MTR-3153, Mitre Corporation, Bedford MA, 1975.

[4] W.E. Boebert and C.T. Ferguson. A partial solution to the discretionary trojan horse problem, *Proceedings of the 8th National Computer Security Conference, National Bureau of Standards*, Gaithersburg, MD, Oct. 1985,

[5] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies, *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, CA, April 1987.

[6] F. Cohen, Computer viruses—theory and experiments. *Proceedings of the 7th DoD/NBS Computer Security Conference*, National Bureau of Standards, Gaithersburg, MD, USA, September 1985, pages 240-263.

[7] D. Denning. *Cryptography and data security*. Addison Wesley, Inc., Reading, MA, USA, 1982.

[8] P.A. Karger. Limiting the damage potential of discretionary trojan horses, *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, CA, April 1987.

[9] E.I. Organick, *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge MA, USA, 1972.

[10] M.A. Schaffer and G. Walsh. LOCK/ix: on implementing UNIX on the LOCK TCB. *Proceedings of the 11th National Computer Security Conference*, 1988.

[11] R.M. Stallman. *GNU Emacs Manual, Sixth Edition, Version 18*. Free Software Foundation, Cambridge, MA, March 1987.

Frozen Files

Frank Kardel
Friedrich Alexander Universität Erlangen-Nürnberg
IMMD IV • Martensstrasse 1 • D-8520 Erlangen • (West-) Germany

1. Problem

At the university of Erlangen-Nürnberg we are running a moderately large network of computers from different vendors running various variants of the Unix[1] operating system. These machines are administered by students and faculty staff of seven faculties. Unfortunately most machines are basically insecure since vendors not always deliver the machines correctly configured in respect to security. Furthermore Unix was not designed to be extremely secure in the first place being a software development system.

The faculty staff cannot secure all machines, so only some relevant machines such as file servers and source machines are taken care off. The remainder, workstations and machines for educational purposes, are basically kept running in a fairly good condition.

The main problem is that it is currently virtually impossible to insure that only trusted users can become the *Superuser* "root". Since "root" is allowed to change anything in a Unix system there is little protection against destruction of files and other manipulations that will for example allow easier re-entry for intruders.

For this reason we needed a mechanism to limit the damage that can be inflicted by "root". The best thing would be a protection scheme that is easy to implement and does not depend on simple mechanisms as user identification numbers (Uid's). Another requirement was that the protection scheme should be compatible with the standard Unix protection scheme in order to allow almost every program to run without change.

2. Frozen Files

Frozen Files is an extension of the standard Unix permission checking system. Files have two mode fields instead of one for determining the rights of a process in respect to a file. Access permissions for a process are checked against the so called "effective mode" which is derived from the "public" and the "private" access mode of the file.

Files can exist in two forms: **normal** or **frozen**. A normal file has only the "public" access mode field which is checked with standard Unix access semantics (normal permissions checking, permissions are ignored for "root" - as usual). A frozen file possesses both the "public" and "private" access modes.

The process of adding the "private" access mode to a file is called freezing. Freezing is done by adding a password to a file. Certain file system operation are forbidden for a frozen file.

These are:

link, unlink, chmod, chown, rename

In order to regain full access to the file the password the file is frozen with must be added to the credentials of the process wishing to manipulate the file. The checking of access rights is changed for frozen files. Processes not in possession of the correct file password are checked against the "public" access mode. Since "root" can change its user id to any value, it can gain access to files where at least one bit is set in the user, group, others field of the "public" mode. So if there is no "write" bit set in any of the user, group, others fields then write access is denied for "root". This semantic allows to effectively deny certain types or even

[1] Unix is a trademark of AT&T Bell Laboratories

all access permissions for "root". If the process has the correct password for the file its access permissions are checked against the logical OR of the "public" and the "private" access mode. Furthermore, the link, unlink, chmod, rename operations are allowed again if the effective access mode permits these operations.

So far, a two level permission checking mechanism has been shown, where "root" cannot break the rules without knowing the correct file passwords. There still remains one more problem with Unix file system semantics.

2.1 Frozen Special Device Files

Special device files allow "root" to do anything since they allow direct access to device drivers and thus raw disc access and on many systems also access to kernel data structures. In order to protect special device files it is not sufficient to protect a single device file entry in the file system because "root" could still create new unprotected special device entries and thus circumvent the protection mechanism. So it is important to protect the device drivers instead of the device files. The frozen file mechanism insures correctly installed special device files by requiring that device files are frozen for protected drivers. This requirement is not enough since "root" could still create a new special device file and freeze it. For this reason another requirement was added: Special device files for protected drivers must be frozen with the same password the system root mount point ("/") is frozen with. The distinction whether a driver is protected or not is made via an additional flag in the *cdevsw* and *bdevsw* structure. There is a need for devices not to be protected by the frozen file mechanism, terminal drivers for example.

2.2 New Rights for Frozen Files

In order to manipulate frozen files a knowledge of a password is required. These passwords are added by a new system call to the process' credentials in encrypted form. It is possible to add several passwords to this list so one can manipulate several differently frozen files. For every password a flag can be set that defines whether the password should be deleted on the execution of the "exec" system call or not.

The s-bit mechanism of Unix finds its counterpart in the p-bit mechanism for frozen files. A frozen file can have the p-bit set. Upon execution of this file the password the file is frozen with is added to the process' password list. This mechanism makes it possible to give programs and scripts additional rights in respect to frozen files.

The p-bit mechanism supports accumulative rights in contrary to the one level s-bit mechanism. Passwords added via p-bits will normally be destroyed on execution of the "exec" system call, unless specified otherwise. This allows interpreters to run with a higher privilege level without the risk of passing password rights to untrusted programs. Another advantage of frozen files is that password rights cannot easily be stolen (saved) by creating p-bit files since the original unencrypted password is required to create the p-bit file.

3. Frozen File Protection Issues

Frozen Files can be used in several ways to increase security. The major advantage is that they are resistent to ordinary "root" users who do not have the correct password. This property allows the installation of programs and configuration files with the advantage, that a "root" user cannot change or move these files, although otherwise having full control over the system.

3.1 Controlled Configurations

It would be advisable to use the frozen file mechanism for protecting the /etc/rc group of files and thus separate the system configuration aspects from administrative "root" user actions.

3.2 Controlled Access

Frozen Files can also limit the privileges given out to certain programs to that, that is needed for the required action.

3.2.1 A Limited Rights Example

A *passwd* program could be frozen with the same password the passwd-file is frozen with. The *passwd* program could then have a p-bit and the private mode of the frozen file is set to be writable by all users, while the public mode of the passwd file is set to be read only. This setup would make the *passwd* program to be the only entity being allowed to change the password- file (except for the Person knowing the passwd file password). The advantage is that not even "root" could not break the rules. So it is possible to build different administrative domains. The danger of rights leaking out of a program can be controlled by the *delete on exec* flag that is associated with each password.

Freezing central system files can also lead improved resistance against worms and viruses.

3.2.2 Frozen Files for Privileges

Due to the possibility of carrying multiple passwords, the Frozen File mechanism is also suitable to build a system of privileges, like the compilation privilege, the system source privilege and alike by using a modified *login* program to set the passwords in the credential structure.

3.2.3 Credential Passwords

Currently a simple mechanism for storing the passwords in the credential structure is used. The password list consists of pairs of facility mask and password value. The facility mask defines the functions the password value can be used for and whether it is to be deleted on the execution of the "exec" system call. Right now only one facility is defined: Frozen Files.

The password list concept a a simple Capability system which can be used for more than just verifying file access permissions. The protection of certain system calls comes into mind and the possibility to encode more rights into the passwords by cryptographic means.

3.3 Device Access Restriction

The possibility of bypassing the protection mechanisms is greatly reduced by having frozen devices. This allows modification of disk data structures and kernel data structures only to a very limited number of users. The protected devices still cannot control the physical access of disks and tape and thus cannot prevent the booting of a suitably modified kernel or direct disk block access, but this mechanism can be hardened by using cryptographic techniques at the *blockio* level.

Though not being able to prevent physical access Frozen Files can be used to harden a system against attacks from the running system, such as patching the kernel or installing a new bootloader.

3.4 Controlled Boot

Since the root filesystem ("/") can be frozen (and must be frozen for protected devices to work) a modified bootloader must be installed on the root filesystem. This bootloader can check the integrity of the operating system image to be loaded by requiring that the operating system is frozen the same way the root directory is frozen with. This requirement insures, that only correctly frozen kernels can be booted. In order to allow booting of alternate kernels a boot can also be permitted, when the user enters the password of the root directory. This mechanism can be circumvented only, if physical device access is present or when the system monitor allows memory modification. Some vendors can deliver "secure" monitor ROMs that will forbid this kind of modifications.

4. Implementation Aspects

Frozen Files do not require much changes. Frozen Files where implemented in SunOS 4.0.3. There have been few changes to some parts of the kernel and some user level programs.

4.1 Complexity

All changes to the kernel where made during a 2 week operating systems project by 12 students who have never done any kernel work before. This shows that required changes are acceptable for the achieved results.

4.2 Frozen File Implementation

The frozen files are implemented as a new file type in the ufs file system. The SunOS vnode-operations have been extended by a new operation that allows the frozen file manipulations. The additional data needed for the frozen files was stored in some unused fields of the disk inode structure. The stat system call also passes the additional data in unused fields of the stat structure. The mode field of the stat structure is set to the effective mode of the file. This allows all programs that have no knowledge about frozen files to run without any compatibility problems.
A few user level program have been extended to allow easy manipulation of the frozen files.

Kernel changes:

- ufs files
  new access mode check, special device support
- new system call
  implementation of password management routines, freeze and thaw operations
- os files
  p-bit semantic for exec system call
  support for new credential structure
- boot files
  new bootloader for frozen files

Changes in user level programs:

- chmod
  allow changes of private mode and setting of p-bits
- csh/sh
  support password addition and deletion
- ls
  display public and private mode of a file (new option)
- fsck
  support frozen files
- find
  allow search for frozen files and specific private modes

5. Summary

Frozen files are only a mechanism and it is yet to be determined how usable they are and whether they will contribute to security. One major advantage is the simple implementation and good compatibility to standard Unix system call interface. Frozen files are currently implemented on a Sun 3 Workstation[2] under SunOS 4.0.3 at the university of Erlangen-Nürnberg.

[2] SunOS, Sun Workstation ®, as well as the word "Sun" followed by a numerical suffix, are trademarks of Sun Microsystems, Incorporated.

Extended Access Control in UNIX System V - ACLs and Context *)

Hermann Strack
European Institute for System Security
University of Karlsruhe

Access control is an important element of security in computer systems. Access control in Standard-UNIX should be improved, considering the different security requirements in the business world and in the public sector. The IT-Security Criteria, edited by the West German Government, specify further requirements for access control in "trusted" systems.

Considering that background, this article describes some conditions for security in UNIX. First there is an general introduction to UNIX Security also on background of Security Criteria, different requirement levels on access control are described, the concept of context related access control is introduced. Then a first realization of enhanced context related access control in UNIX V is proposed as well as some possible further developments.

---

*) UNIX is a registered Trademark of AT&T
This article is a worked out version of one published on the First German Conference about Computer Security, May 1990

Author: Dipl.-Math. Hermann Strack,
European Institute for System Security (E.I.S.S.)
Universität Karlsruhe, Postfach 6980,
D-7500 Karlsruhe,
West.-Germany,
Tel.: +49- 721 / 608 4025 or 608 4304
Fax : +49-721 / 69 68 93
Mail (EUnet): STRACK@IRA.UKA.DE

Overview

Two of the main development lines of UNIX are AT&T's UNIX System V and BSD-UNIX of UC-Berkley. Other descendants are offered by a variety of makers. The interfaces of UNIX System V are specified in the System V Interface Definition (SVID) of AT&T. Standardization efforts around UNIX are taken at the IEEE (POSIX) and X/OPEN. Also, release 4.0 of UNIX System V of the source licenser AT&T is to rejoin the major UNIX lines upward compatibly (cf. (7) Martin).

Securitywise, UNIX V 4.1 is supposed to reach evaluation at B2-Level of the Orange Book (TCSEC) in the U.S. (cf. (7) Martin). AT&T offers the UNIX V/MLS, which has been evaluated at B1-Level of the Orange Book (B1 cf. next paragraph).
From research institutions: The German National Research Center For Computer Science (GMD) St.Augustin/FRG, presented "Birlix" in spring of 1990, which is a distributed operating system, offering essential security features and also a BSD-UNIX 4.3 emulator (cf. (6) Kowalski).

Security has certainly not been a central aspect for the design of UNIX. However existing mechanisms can be enhanced easily and compatibly to match commercial needs. Considering different levels of requirements, this article will describe the conception and realization of a first UNIX V prototype with enhanced access control, whose design was guided by the author at Mannesmann Kienzle GmbH Villingen/FRG. Proposals will be made for further developments. To not exceed the scope of this article, only selected aspects of access control are covered.

Access control, meaning the administration and checking of rights, is one of the essential basic functions to cover security demands. Others are identification/authentication, audit, object reuse, error recovery, continuity of service and data communication security (cf. (13) ZSI).
To guarantee authenticity in communication (man-machine, machine-machine), the European Institute for System Security supplies methods, which can also be adapted for UNIX.

An attractive design feature of UNIX, which is also a source of incompatibility and security problems, is the adaptable UNIX user-interface. On the one hand, this adaptability is offered for administration (e.g. user specific definitions for session-interfaces in /etc/passwd). On the other hand, many system functions are outside of the UNIX kernel as utility programs and can therefore be replaced by other programs with additional features, even without changing the UNIX kernel by putting them on top of the systemcall-interface. So it is possible to create new mechanisms and interfaces also for access control, yet with hidden usage of critical UNIX mechanisms such as the superuser-sbit programs. If however, for some users there is realistically an escape option back to the original user-interface or access to software production tools (e.g. in order to keep up compatibility and to retain software investments), the same problems occur as in the original UNIX. The extendability of the UNIX system should be looked upon as independent from offering adequate security functionality of a UNIX base system.

Some selected problems of access control in UNIX V base systems from a user's point of view:
- insufficient restriction and control of superuser access (realization is possible through chroot()-call and special superuser interfaces, protected logging)

- insufficient granularity and missing possibility to link rights to contexts (e.g. to access programs); as a substitute sbit/setuid programs
- process or program bound restrictions on access area can not be parametrically controlled
- universal flexibility of UNIX versus restrictability, error and manipulation robustness (insufficient capsulation of data, insufficient parameter check)
- problem oriented documentation especially for programmers (like X/OPEN Security Guide; in (12)), user support through tools, overlookable mechanisms, active and passive help systems and the use of appropriate representation techniques, also at a semantic level, should all not be missing in the security sector.

The administration of access control is based on knowledge about structures and organizations in real life and about their representations on computer systems. Today´s computer systems like the UNIX base system generally offer representation of knowledge only in a implicit way through procedural representations. The descriptive aspect is neglected. Improving man-machine communication through the methods of knowledge processing is a matter of research (cf. (8) Riekert).
Research domains with major security relevance are formal specification and verification at system development stage - UNIX was not developed under such conditions.

"Trusted" IT-Systems

The grown and further increasing interest in aspects of the topic computer security in modern societies documents the great and ever growing importance that the use of computers has gained in these societies. This upgrading is not only expressed through increased attention of the media towards topics in electronic data processing in general and aspects of computer security specifically, but is also shown by activities of national legislators (e.g. : USA , FRG).

These activities are on the one hand meant to bind electronic data processing closer to legal order (e.g. data protection, copyrights and criminal law), and on the other hand also deal with more technical questions about standards and classification of information technology systems (IT-Systems), specifically in the field of "trusted" IT-Systems. Basic documents on the latter are the so called Orange Book (Department of Defense (DoD): Trusted Computer System Evaluation Criteria, USA 1983) and its successors, and recently the IT-Security Criteria (German Information Security Agency (ZSI/GISA): IT-Security Criteria, Criteria for the Evaluation of Trustworthiness of Information Technology (IT) Systems, FRG 1989). Harmonized Security Criteria edited by four European Countries will appear at Sept. 1990 (cf. (3), (4a), (13)).
Both documents the Orange Book and the IT-Security Criteria cover specifically "trusted" systems instead of "secure" systems. Only a overall view of different interconnected aspects of environment-user-system relationships, like user demands, the application environment or possible threat, make it feasible to talk about security. In that way defined and also evaluated security with all its complex relativity is more precisely characterized as "trustworthiness".
The Orange Book includes a catalogue of requirements, which serves to evaluate a computer system according to its "trustworthiness". At the evaluation, the systems are put into one of 4 possible

classes D, C, B or A (the classes B and C have further subdivisions). The ordering D, C, B, A translates to an increase in security functionality and complexity of evaluation. The classes are strictly hierarchic. Every class meets the requirements of its predecessors, where class A is the highest class (cf. figure 1.).

DAC:
Division D : Minimal Protection
Division C : Discretionary Protection
C1: Discretionary Security Protection
C2: Controlled Access Protection

MAC:
Division B : Mandatory Protection
B1: Labeled Security Protection
B2: Structured Protection
B3: Security Domains
Division A : Verified Protection

Figure 1: Orange Book with two access policies

Although the Orange Book is published by DoD, it is the U.S. standard for military and civilian sectors. Starting 1992, all computer systems in the U.S. public sector, which contain or process classified or sensitive non-classified information, should at least be evaluated at class C2 (cf. (9) Stobbe). Evaluation and certification of computer systems according to the Orange Book will be done in the U.S. by the NCSC (National Computer Security Center, formerly DoD Computer Security Center), however only for U.S. manufacturers.

In the Federal Republic of Germany, the IT-Security Criteria were developed by the ZfCh/GCB (Zentralstelle für Chiffrierwesen/German Cipher Board, predecessor of the ZSI/GISA) with the support of public agencies, computer related industry, various industrial lobby federations, research and university institutions. The Orange Book was used as a reference. However the IT-Security Criteria allow a more distinguished and thorough view of "trustworthiness" than the Orange Book. Since the Orange Book can't judge the three aspects "functionality", "assurance" and "evaluation complexity" independently, the IT-Security Criteria offer a judgment of the "trustworthiness" of IT systems, treating the independent dimensions "functionality" and "assurance" separately. "assurance" and "evaluation complexity" are not distinguished, since in practical applications these come out to be the same (cf. (5) Kersten). The IT-Security Criteria allow independent setup of an functionality profile and an assurance profile for system evaluation, via a selection of base functions and assurance levels relative to application specific requirements. This is not possible using the Orange Book. Producers of computer systems and their customers are offered a greater chance to respond to their needs flexibly and adequately using classification according to the IT-Security Criteria instead of the Orange Book.

The IT-Security Criteria offer functionality classes F1 to F10 and assurance levels Q0 to Q7 for a comparative judgement. The possibility of adding further functionality classes was explicitly included. Generally, useful recommendations can be added. While all assurance levels are ordered hierarchically up to the highest level Q7, only the functionality classes F1 to F5 are ordered. They are defined to correspond to the functionality definitions of C1, C2, B1, B2 and B3/A1 in the Orange Book (cf. figure 2). That way it is possible to fit products, which have been evaluated according to the IT-Security Criteria at functionality classes F1 to F5, directly into the system of the Orange Book (cf. figure 3). A reverse mapping is not possible without further investigation. For that, "assurance" is not distinguished well enough in Orange

Book classes (cf. (5) Kersten).

Consider the original literature (cf. (3) DoD, (13) ZSI) for more details on the Orange Book or the IT-Security Criteria. Questions on access control are discussed on the basis of these two documents in the second chapter.

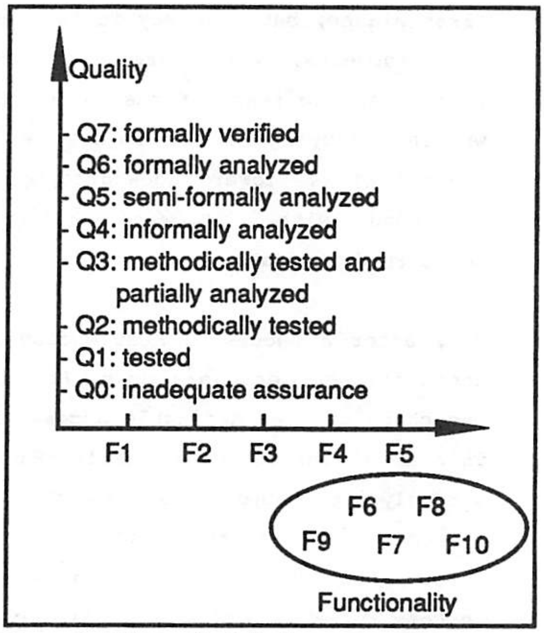
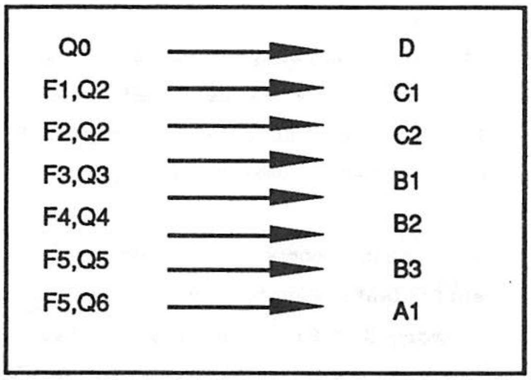


Figure 2: IT-Security Criteria



Q0	D
F1,Q2	C1
F2,Q2	C2
F3,Q3	B1
F4,Q4	B2
F5,Q5	B3
F5,Q6	A1

Figure 3: Mapping IT-Security Criteria → Orange Book

Associations like the IEEE, X/OPEN or USENIX, /usr/group work on UNIX security to include security relevant requirements into their standards (like POSIX) and documents (cf. (12) X/OPEN Security Guide).

The topic "security in computer systems" includes many aspects. Besides to system related ones, one has to consider those related to the system's application environment (e.g. the organizational, personal, technical and even architectural setting). To clarify some terms, a few quotations are made from the IT-Security Criteria now: (cf. (13) p.5).

"The first consideration for the formulation of the security policy for an IT system is always the threats to which a system is exposed. The threats are a function of the operational environment of the system and the sensitivity of the data being processed. The three basic threats to which an IT system is exposed are:

- unauthorized access to information (loss of confidentiality)
- unauthorized modification of information (loss of integrity)
- unauthorized impairment of functionality (loss of availability)

These threats can have vastly differing significance for different IT systems...

... Now part of these threats can be reduced by measures applied within the operational environment of the IT system so that these threats are no longer effective... The remaining threat may then be dealt with by technical measures within the IT system. Hence this remaining threat is a function of the operational requirements of the system, the sensitivity of the data processed in the system and the operational environment, including the technical or organizational protection mechanisms installed there. However this means that the remaining threat also generally differs broadly from system to system. The security policy and the resulting security functions in the IT system itself should then also differ accordingly." (End of quotation)

Use of **access control** is a means to counter the remaining threat to the system. The basic functions "administration of rights" and "verification of rights" cover the aspect of access control in its narrow sense as it is used in the IT-Security Criteria.

Access Control in "trusted" IT-Systems

For the administration of rights both the Orange Book and the IT-Security Criteria distinguish two different policies (cf. figure 1):

- user definable access control **(Discretionary Access Control,** short **DAC)**
- prescribed, rulebased access control **(Mandatory Access Control,** short **MAC)**.

With user definable access control on objects, the authorized user, who is usually only the object´s owner and a privileged administrator, can freely decide to grant or revoke rights on the object. On an occasion of an access request the given rights are checked and the access is allowed or not.

A different understanding of access control comes into effect with mandatory access control. Here the control of information flow is the dominant paradigm (Protection of "confidentiality"). Roughly spoken objects (information) are classified according to certain levels/categories. Subjects (e.g. users) are cleared for certain levels/categories. A subject can only access an object, if his authorization is sufficient for the level/category of the object, which is being decided on by a rulebase. In particular the following rules are valid for reading and writing access:

- The subject can only access an object for reading, if its authorization is at least as "high" as the object´s level (no read-up).
- The subject can only access an object for writing, if its authorization is at most as "high" as the object´s level (no write-down).

The second rule may be quite stunning at first glance, but that way it is prevented that subjects, which are authorized to read, pass the read information on through writing (copying) to an object, which is classified as "lower" (downgrading). Then a subject with a "lower" authorization could read the data.

Only after a successful, rulebased access check further tests are done. For example the check of user definable access rights. This model introduced by Bell-LaPadula is strongly oriented towards the usual treatment of secret and confidential information in military and executive sectors (Setup of security classes, authorization for those). Integrity and availability aspects are neglected.

Figure 1 already depicts, which access control policy is demanded by the Orange Book or the functionality classes F1 to F5 of the IT-Security Criteria.

For most commercial sectors, DAC is sufficient. Especially, if in addition to C2 more DAC functionality and assurance is included according to the IT-Security Criteria. Requirements for the highest possible DAC classes without MAC, which are F2 or C2 resp., are quoted in part from the IT-Security Criteria (cf. (13) ZSI p. 28/29):

"Administration of Rights

The system shall be able to distinguish and administer access rights between users and/or user groups and objects subject to the administration of rights. It shall be possible to deny users or user groups the access to an object completely and to restrict access to non-modifying access. It shall be possible to grant the access right to an object down to the granularity of a single user. Granting and revoking of access rights to an object shall only be possible by authorized users. The administration of rights shall provide controls to limit the propagation of access rights. In the same way the introduction of new users and the deletion or suspension of users shall be possible only through authorized users.

Verification of Rights

With each attempt by users or user groups to access objects subject to the administration of rights, the system shall verify the validity of the request. Unauthorized access attempts shall be rejected.

Audit

The system shall contain an audit component which is able to log the following events with the following data:
...Attempted access to an object which is subject to the administration of rights:
Data: Date; time; user identity; name of the object; type of access attempt; success or failure of the attempt.
Creation or deletion of an object subject to the administration of rights:
Data: Date; time; user identity; name of the object; type of action..."
(End of quotation)

The comparision of the UNIX System V up to release 3.2 with these requirements shows that enhancements are necessary for the individualization of the distribution of rights and for the logging of access attempts. Individualization is only possible at a granularity of owner-group-others in UNIX V. It is not possible to assign separate rights for users other than the owner or user groups other than the objects´s group. A thorough logging of access attempts is not offered.

UNIX was not designed to handle most sensitive information under maximum threat in a secure and also simple comfortable manner. Therefore possible solutions based on UNIX V access control ask too much of the system´s user, because of its complexity, insufficient "fault tolerance" (missing redundancy and operation typing of access control) and insufficient user support. Thus there is a source of security gaps. With relatively small extensions, UNIX can be improved to become a system with decent "trustworthiness" though, for dealing with sensitive non-classified information.

Application Oriented Access Control

In every day life, rights are generally linked to the corresponding context:
- A car may only be driven on public roads, if the driver has a drivers license, he is in proper physical condition, the car is in a prescribed shape and so on. The elementary authorization (drivers license) alone is not sufficent (The right is linked to multiple contexts).
- Companies and public institutions don´t decide on payment or purchases without a well founded request, recommendations for the request, approval of the request and a payment order signed by certain authorized persons. For other decisions there are cases, where approval of a hierarchy chain is necessary. In some cases the responsibilities are delegated and in others a decision can only be reached through agreement of equally entitled

parts (e.g. different departments). (Aspects: reference to contexts, shared control, hierarchy and agreement schemes, delegation and capsuling of responsibilities).

Simple layer models for confidentiality and integrity certainly don´t capture all aspects of such differently segmented access realities especially in commercial environments (cf. (2c), (7a)).

The orientation towards more computer support for all activities in corporate life results in ever more access decisions being made inside of these systems for most complex contexts. The capability of systems to handle those access decisions is becomming an increasing factor for use of these systems at all. A reference to contexts must therefore be possible for access control.

Essential aspects for the definition and administration of context related access rights are (keyword ACCESS-REQUIREMENTS):

- simple and uniform use
- flexibility with respect to the definition of temporary rights
- flexibility and granularity of the definition of rights and contexts
- support of grown organizational structures
- Need-to-Do: only grant as many rights as needed
- inherent fault tolerance with respect to handling
- security of the mechanisms against circumvention (especially of access contexts).

Additional requirements result from those problems named in the "overview" chapter - especially the need for access control based on the concept of abstract datatypes. An example for a missing single feature in UNIX V: a separate right for link/delete.

Context and Access Control in UNIX

UNIX essentially offers two possibilities to include contexts at granting of rights:

1. Transfer of ownership of the accessed object to a privileged user and the use of the group mechanism.

2. Transfer of ownership of the accessed object to a privileged (pseudo) user/group and writing of special access programs with use of the sbit/setuid/setgid mechanism (effective change of user or group resp.).

1.) :

Assume for example that a path to a file, which a user tries to access, contains a directory that doesn´t belong to him and which he can´t delete. Now it can be achieved that the user can only access the file if he is/was effectively in the group of that particular directory at the time of access. This is done by withdrawing the x-others bit. Thus the access to central data within a directory can be related to a context (mapping of context to group membership). The authorized users must be admitted to the group by the administrator (Entry in the file /etc/group) and have to enter this group every time before an access e.g. with the "newgrp" command.

2.) :

Through s-bit mechanisms in combination with setuid/setgid systemcalls it is possible to run special access programs with the effective user or group ID of another user (in general a more privileged one). That way the temporarily aquired user/group ID´s rights of the access programs become effective instead of the real rights of the calling user.

How can these possibilities be judged with respect to what has been demanded at access control above (keyword ACCESS REQUIREMENTS) ?

at 1.) :

This loose treatment of context binding (the access operations are not specified) will suffice in many cases. The granting of rights is complex in use though: The administrator has to create a new group and place the authorized users in this group. If the objects of concern are already existing, they must be transfered from their owners to the new group with the "chgrp" command. The latter violates the former organizational structure though, since the members of the old group can´t access the files anymore. This can be fixed by including the members of the old group into the newly created one. That however has the negative effect that those members of the old group can also access all files of the new group, what might be contrary to the initial intent. Obviously there is no flexibility nor ease in use. Not even the support of former organizational structures or the Need-to-Do principle can be guaranteed in all cases. A duplication of data or a use of the second possibility might be inevitable.

The administrator has a problem: He has to know and account for the intention of every group. For a larger number of groups, the administration remains overlookable only if there is documentation about the functions and set theoretic relations of groups (it is not even possible to include comments in /etc/groups though). The problem of handling just a few groups with direct regard to their organizational setting may be an acceptable one. But as soon as an excessive use of "constructed" groups (as a substitute for missing subject granularity in the definition of rights in UNIX V) is made, this approach is no longer practical. Sometimes a sufficient subject granularity for the definition of rights is argued for by stating that K users can only make up 2^K-1 different groups. That shows little understanding of the problem in practice though. Especially for those "constructed" groups, which are usually not related to organizational units, it is annoying that after a "newgrp" command one can´t access the objects of the other groups anymore (until another "newgrp"). This is caused by the fact that in UNIX V 3.2 objects and subjects can only be associated with one group and one owner at a time.

at 2.) :

The problem of writing s-bit/setuid/setgid programms is well known (cf. Bishop (2a), Bunch (2b)). Such programs, especially setuid-superuser programs have to be written very carefully to prevent the complete security collapse in UNIX systems (aqusition of complete superuser privileges).

Special access programs, which use those mechanisms, must therefore be written by the system´s maker and must be so general that user requirements are configurable. This could for example be done by taking access lists into account before the actual access.

It must be avoided that the writing of such programs is left to the commercial user. With the help of such programs, a few requirements could be met. The basic disadvantages of sbit/setuid/setgid programs remain though :

- rights on access operations are not defined and represented explicitly at the object (or at the access subject). The object´s owner cannot check, which access routines exist for the object.
- sbit/setuid/setgid programs jeopardize the Need-to-Do principle. Instead of granting of demanded rights on access operations they allow temporary acquisition of substitute user/group identities, which are again restricted by source code to certain objects/operations (hopefully, yet not simply verifiable by

the user). A other problem with the use of such special access programs is the integration of existing UNIX software. It usually accesses files through direct systemcalls without calling separate processes. Migration effords would be necessary. Also a loss of performance would have to be accepted.
Other access control problems were mentioned in the "overview" chapter.

ACLs in UNIX V - an approach at Mannesmann Kienzle

A first prototype of an UNIX V 3.2 Kernel with extended access control exists. The following aspects were considered at system development: (keyword UNIX-ACCESS-SYSTEM):

- at development stage: use of existing mechanisms, minimum change of the kernel, portability
- compatibility to UNIX V 3.2
- simple and uniform handling
- flexibility with respect to the definition of temporary rights
- flexibility and granularity of the definition of rights and contexts
- support of grown organizational structures
- distribution of rights according to user competences and roles
- Need-to-Do: only grant as many rights as necessary
- inherent fault-tolerance with respect to handling and systems operation
- security against circumvention

The objects of the standard access control in UNIX V 3.2 are file system objects as well as objects of process communication (Message queues, semaphores, shared memory sectors).
The requirements of context consideration, flexibility and granularity of distribution of rights and of context definition, support of grown organizational structures and the Need-to-Do principle lead to the following structure of an enhanced definition of rights:

For every object there is an access control list (ACL), in which the elementary rights are defined dependent on the user and context:

```
user_1   context_1   right_a
user_1   context_2   right_b
user_2   context_2   right_c
user_2   context_3   right_d
```

or in short as a mathematical function:

```
rights = acl_func ( user, context )
```

The context will in general consist of a number of context columns or parameters resp.. At the moment we consider the following contexts:

group, access program, access time definition

or in short:

```
rights = acl_func (   user, group,
                      access_program,
                      time_definition )
```

Rights can also be non-rights. This means refused rights.

Now if there is an access attempt, the rights are checked by verifying whether the user in his current context matches one line or multiple lines of the object´s ACL. In the case of matching, the ACL right definition belonging to the user and his context is checked versus the desired right. Otherwise (no matching ACL lines) a DEFAULT action is done. Thus possible results of the check are:
"Yes", "No" or "DEFAULT".

The result of acl check in short notation:

```
check_result =
acl_check ( desired_right,
            acl_func ( calling_user,
                       current_context )
          ).
```

Through the access program context in combination with the group context, the s-bit/setuid/setgid programs as means of access context representation in standard UNIX V become obsolete (exception: superuser systemcalls for non-superuser). Now the rights are defined at the objects.

The next decision to be made is how to use the result of checking. Specifically, the question is how the cooperation of ACLs and the normal UNIX rights checking looks like. The following setups could be feasible (cf. Sutton (10/10a)):

- The ACL system replaces the old UNIX access system completely
- ACLs optionally instead of UNIX access bits: every object has either an ACL or standard UNIX rights
- ACLs optionally in addition to UNIX access bits: cooperation of ACL check with UNIX rights check for every object

Since the granularity of UNIX rights bits is sufficient in most cases, the last option was selected for compatibility reasons. ACLs can be created for every object. ACLs are supposed to make circumvention of the rights definition impossible. Therefore ACL rights definitions always overrule UNIX rights bits. If the ACL does not make any statement however, the DEFAULT action is the standard UNIX rights check (see above). For reasons of security against circumvention, the real user ID is used for the check instead of the effective one (only the superuser can change the real user-ID).

ACLs should be used for almost all parts of the administration of rights (e.g. administration of menu objects, spooler queues or any freely defined objects). Therefore the ACL interfaces are available for these freely defined objects and for UNIX objects in the same manner, both for command- and C-Level.

The following basic functions are offered to handle ACLs:

acl_syscr	-	create ACL catalogue
acl_sysdel	-	delete ACL catalogue
acl_exist	-	list the ACL objects of a set of objects
acl_new	-	create or overwrite an ACL
acl_list	-	list the contents of an ACL
acl_delete	-	delete an ACL
acl_iin	-	update or create an ACL
acl_iout	-	delete single contents of an ACL
acl_ivalid	-	check for matching: user+context+rights versus ACL lines
acl_iallows	-	given user+context, it returns the elementary rights

ACLs and ACL catalogues can only be accessed using these "trusted" basic functions by a normal user. The current rights control policy: changes in an ACL of an object can only be made by the objects owner or by the administrator. Every change in the ACLs and every check is logged in protected files.

Different policies for access to ACLs can be achieved by defining ACLs for the ACL catalogues themselves by the ACL-Administrator or by use of ACLs for freely defined objects in new ACL commands. The hierarchical structure of the file system becomes in effect for access at ACLs of file system objects as it is in UNIX V:

Reading access to an ACL of an object is only possible, if all the directories above the object are accessable (x-bit) and the directory above the object is readable.

As mentioned above the basic functions exist both as program interfaces and as commands. The commands can be used interactivly or in batch mode. Except for acl_syscr/acl_sysdel, all commands generate outputs in a simple control file format (cf. figure 4.). This format can then be the input to other commands (e.g. acl_exist, acl_list, acl_new). These control files support the administration of rights concerning:

- documentation of the structure of rights
- comparision of actual with intended rights
- description of the rights for organizational structures
- software delivery (definition of user roles/competences by the producer)
- software installation.

For a first prototype of an ACL system, the functionality was reduced: ACLs are applicable to file system objects and to freely definable objects. The only context column is one for the group context.

Flexibility in the use of ACLs is important for their practical application. Short-term changes in the ACLs and default setups should be supported. The prototype offers possibilities as follows (cf. figures 4./5.) :

```
# comment: this is an example
# for an ACL control file with use of
# simple wildcards;
# white space character: +
@ACL:  FILE
@CWD:  /usr/finance

# ACL-objects ( some relative to @CWD ):
            license/receipt
            license/expense
            invoices
            /usr/book_keeping
            (

                 miller      book      r+w+Nx

# substitute: smith for baker
                 smith       book      r+w+Nx


# commented out because of vacation:
            ^^^  baker       book      r+w+Nx
# invoice controlling and warning:
                 NN          warn      r++Nw+Nx
# others:
                 NN          NN        Nr+Nw+Nx
            )
# end of first ACL
```

Figure 4: an ACL control file with simple wildcards

1. Through two classes of wildcards, priorities can be defined on ACL-lines: High priority wildcards ALLUSER / ALLGROUP always beat explicit naming of users or groups and can therefore easily be used to temporarily supersede present ACL definitions without destroying them. Simple wildcards NN will always be beaten by explicit naming. This way defaults can be defined quite simply.
2. Lines of an ACL can be commented out temporarily (no actual deletion).
3. Comments can be placed anywhere in ACL control files.
4. If a rights definition is incomplete in one line, other lines of the ACL are consulted.

```
# comment: this is an example
# for an ACL control file with use of
# different wildcards;
# white space character: +

@ACL:  FILE
@CWD:  /usr/finance

# ACL-objects ( some relative to @CWD ):
          license/receipt
          license/expense
          invoices
          /usr/book_keeping
          (
# revision:
               ALLUSER book      Nr+Nw+Nx
               ALLUSER revision  r++Nw+Nx
                  miller      book     r+w+Nx

# substitute: smith for baker
                  smith       book     r+w+Nx

# commented out because of vacation:
             ^^^  baker       book     r+w+Nx
# invoice controlling and warning:
                  NN          warn     r++Nw+Nx
# others:
                  NN          NN       Nr+Nw+Nx
       )
# end of first ACL
```

Figure 5: ACL control file with different wild cards

The ACL-check for file system objects was integrated into the access control mechanisms of the UNIX kernel. That way the highest possible security against circumvention was reached and existing UNIX software could be included into ACL functionality. To minimize development cost and to keep within the UNIX kernel philosophy of "small is beautiful", only the essential basic functions were put in the kernel. The ACL-change functions were realized at process level.
For every access attempt to a file system object through systemcalls, the ACLs are consulted before the normal UNIX rights check. If access is denied by the ACL, the systemcall is ended with the error errno=EACCES (UNIX V compatible). Otherwise, if the demanded right was included in the ACL definition, the systemcall will proceed as normal. If nothing was stated in the ACL, the DEFAULT action is the standard UNIX rights check.

Review

Our approach was to modify the UNIX access system according to the aspects mentioned above at keyword "UNIX-ACCESS-SYSTEM". Considering these aspects, a significant improvement compared to standard UNIX was achieved. The completion of the prototype through enhanced access contexts will offer the user uniform application-, context-, object- and organization oriented administration of rights.

The solution is compatible to existing UNIX software. Some programs, which have an own administration of rights through stat/fstat systemcalls, might have to be adapted. However that can be avoided in many cases by placing ACLs on intermediate directories.

The main requirements of functionality class F2 of the IT-Security Criteria at "access control" are claimed to be met.

Extensions and Further Developments

Granularity and control of rights:
ACLs for file system objects can be enhanced fairly easily by a link/delete right. Different strategies are possible to introduce further control policies for the rights distribution (mandatory control beyond Bell-LaPadula, restrictions for administrators, shared control with hierarchy and agreement schemes). Especially it should be possible to separate completely the use of control rights (meta/administration rights) and the use of the controlled rights itself (e.g. mandatory control of process environment). This problem is strongly related to the problem of superuser control in UNIX V.

Program/process restrictions:

Through an introduction of PACLs it should be possible to restrict/map the access areas of programs/processes (e.g. to a subtree of a file/process tree). Such definitions of rights should be devided into a program owner part and an administrator part, which controls the owner part. By this way it is possible to control sbit/setuid/ setgid-programs in an effective manner with a clear representation of control mechanism to outside.

Granularity of contexts:

Access history could be a important context in some cases. A free ACL context should exist as an additional context to restrict access programs. The interpretation of this free context is done by the access program in a documented and to outside represented manner. In all cases with context access program, the access program has to be protected implicitly against changes by those authorized to call it.

Datatyping on Executables

To represent compiled executables by simple UNIX files, whose contents can be modified after creation, is not a good idea from a security point of view. There should be a function to create executables as an unmodifiable unit.

Privacy and Liability - Superuser and ACL administrator restrictions:

The principles of privacy and liability (e.g. authenticity, confidence, integrity, proofability) are jeopardized by the superuser feature of UNIX and other operating systems. This is an very serious handicap for use of such systems especially in commercial environments. To improve something here means perhaps to make a new system - but I think improvements inside UNIX are possible, too. First of all - to make superuser actions liable and proofable there should be a protected kernel administration and then a protected logging by the kernel. Protection is possible by combination of organizational, hardware and software methods including cryptografic ones (digital signatures, encoding, shared control/keys, WORM storage).

Even without changing the kernel it is possible to realize different superuser interfaces with different logins and strongly restricted possibilities to call and control programs/processes (substitution of standard command interpreters). A defined and validated program environment must be created for superusers or other privileged users.

ACLs definitions already affect superusers in the prototype kernel. They can be bypassed by superuser only by redefinition through switching into the ACL administrator's role (su-command) or modifying storage devices or kernel resp. ACL software. If ACL redefinition by superuser could be prevented, through ACLs with context access program it should be possible to protect the devices and the software against superusers, if the access programs are validated and "trustworthy". Similarly the ACL administrator must be restricted, so that he cannot cumulate rights on operations by using his control rights (meta rights, see above). Further research is necessary here.

Representation techniques:

Comments should be allowed anywhere in the ACLs as well. There should be semantic links between the ACLs, which prohibit independent changes in these ACLs. Techniques to visualize configurations and rights and their semantics should be developed. Object oriented knowledge processing might be the key to methods for these purposes.

Literature:

(1) Bach, Maurice : The Design of the UNIX Operating System; Englewood Cliffs, New Yersey, USA 1986; Prentice Hall

(2) Beth, T.; Gollmann, D. : Formale Methoden: Korrektheitsbeweise und praktische Sicherheit; E.I.S.S. Report 90/3, Universität Karlsruhe FRG

(2a) Bishop, M. : How To Write a Setuid Program; ;login: The USENIX Association Newsletter, Vol. 12(1), 1987

(2b) Bunch,S. : The Setuid Feature in UNIX and Security; National Computer Security Conference, 1987

(2c) Clark, D.D.; Wilson D.R. : A Comparison of Commercial and Military Security Policies; in: Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland California

(3) Department of Defense : Trusted Computer System Evaluation Criteria CSC-STD-001-83; Fort George G. Meade, Maryland USA 1983

(4) Gleißner; Grimm; Herda; Isselhorst : Manipulation in Rechnern und Netzen; Bonn FRG, 1989, Addison-Wesley

(4a) Governments of France, Germany, the Netherlands, the United Kingdom : Information Technology Security Evaluation Criteria (ITSEC); Harmonized Criteria of France - Germany - the Netherlands - the United Kingdom, DRAFT, Bonn May 1990, final version to appear at ISEC Conference Sept. 25/27, Brussels 1990, arranged by the Comission of the European Communities

(4b) Groß, Michael : Vertrauenswürdiges Booten als Grundlage authentischer Basissysteme; to appear by German National Research Center For Computer Science (GMD), Darmstadt FRG, 1990

(5) Kersten, Heinrich : Der deutsche Kriterienkatalog für die Bewertung von Sicherheit und Vertrauenswürdigkeit von DV-Anlagen; in: Recht der Datenverarbeitung, Heft 1 1989; Datakontext Verlag Köln FRG

(6) Kowalski, Oliver C. : Schutz im BirliX-Betriebssystem; Konzepte und Implementierung; Diplomarbeit Universität Bonn FRG, 1989

(7) Martin, Günter : UNIX System V, Release 4.0: Die Integration bisheriger UNIX-Varianten; Tagungsband der GUUG-Jahrestagung 1989, NETWORK GmbH FRG

(7a) Muffet, J.D.; Morris S.S. : The Source of Authority for Commercial Access Control; in: Computer, Feb.1988, Vol. 21 (2),IEEE

(8) Riekert, Wolf-Fritz,J. : Werkzeuge und Systeme zur Unterstützung des Erwerbs und der objektorientierten Modellierung von Wissen; Dissertation Universität Stuttgart FRG, 1986

(9) Stobbe, Christine Stöcker, Elmar : Software - Sicherheit und Bewertung; in: Informationstechnik it, Heft 1 1990, R. Oldenbourg Verlag FRG

(10) Sutton, Steve : Can UNIX be trusted?; in: CommUNIXations Sept./Okt. 1987; /usr/group Santa Clara USA

(10a) Sutton, Steve : Trusting UNIX; in : CommUNIXations May/June 1989; /usr/group Santa Clara USA

(11) Wacker, Birgit : Zugriffsschutz unter UNIX; Diplomarbeit Fachhochschule Furtwangen, 1988, FRG

(12) X/OPEN Company, Ltd. : X/OPEN Security Guide; Prentice Hall 1989

(13) Zentralstelle für Sicherheit in der Informationstechnik ZSI : IT-Sicherheitskriterien, Kriterien für die Bewertung der Sicherheit von Systemen der Informationstechnik (IT); Köln FRG, 1989; Bundesanzeiger Verlagsges.mbH

How Crackers Crack Passwords
or
What Passwords to Avoid

Ana Maria De Alvaré
Lawrence Livermore National Laboratory

Abstract

Computer security is a growing concern in research, development, marketing, and most other areas of everyday life. The first and foremost task in computer security is to prevent unauthorized access to systems. This report tells how "crackers" (computer wizards who use their talents for illegal and destructive purposes) obtain access to computer systems and gives specific advice on how to prevent them from doing it.

Introduction

Every system manager has a personal set of rules and a checklist to make his or her system secure. Usually those rules include guarding against illegal logins, files, programs, and privileged users. Also, the rules should help preserve the confidentiality of certain data, watch over modem usage, prevent illegal file transfer, and minimize opportunities for break-ins. To prevent a system from being used illegally, passwords are given to all users for their own accounts. Each password is an authorization tool to let the user enter the system. Passwords should be difficult to guess, easy to remember, changed frequently, and protected by the system [JC 84].

Lists of requirements or guidelines to satisfy those elements have been written in textbooks, security handbooks, and journals [KM 79, KW 94, MT 79, NC 85, NN 88, RB 84, RB 87, WK 87]. Rules for password distribution, creation, usage, and storage have been published [HH 84].

In the recent literature have been suggestions about what kind of password not to use and what kinds are secure and/or user friendly (i.e., easy to remember). Tables 1 through 4 contain examples of easy-to-memorize and user-friendly passwords. Some of the suggestions are supported by mathematical proofs, while others have been supported by language and speech studies and by cognitive science [BB 84, KM 79, MT 79, NC 85].

Table 1
Illustrative sequence of passwords associated with the lines of a chosen childhood verse.

Verse Line	Password
1. One for the money	14munny
2. Two for the show	24show
3. Three to get ready	32ready
4. Four to go (to)	42goto

Table 2
Summary of relations among chosen names of cities, intermediate expressions and selected passwords.

City	Intermediate Expression	Password
1. Paris	I Love Paris in the Springtime	ILPITST
2. Rome	Three (Bright) Coins in the (Trevi) Fountain	TBCITTF
3. New York	The Sidewalks of New York (City)	TSWONYC
4. San Francisco	I Left My Heart In San Francisco	ILMHISF

Table 3
Relations among foods disliked during childhood and selected passwords.

Expression	Password
chocolate-covered peanuts	chocovpea
pepsi-cola (and) pretzels	pepcolpre
pineapple-coconut suckers	pincocsuc
fried (-) eggplant	frieggpla

From Barton and Barton[BB 84]

Table 4
Transform techniques to generate passwords based on a chosen expression.

Transform	Illustrative Expression	Resultant Password
1. Transliteration	Photographic Schizophrenic	fotografik skitsofrenik
2. Interweaving of characters in successive words (or numbers)	duke, iron tent pole	diurkoen tepontle
3. Translation	strangers	etranieri
4. Replacement of letter by decimal digit (modulo 10 index of letter in natural order)	cabbage	3122175
5. Replacement of decimal number by letter (with corresponding position, in natural orders)	10/ 12/ 1492	jabadib
6. Shift from "home" position on keyboard.	zucchini	xivvjomo
7. Substitution of Synonyms	coffee break	javarest
8. Substitution of Antonyms	stoplight	startdark
9. Actuation of keyboard "shift"	6/ 6/ 1944	^?^?!($$
10. Substituion of Abbreviations	relative humidity	relhum
11. Use of Acronyms	Mother Against Drunk drivers, Nat'l. Orgn. of Women	maddnow
12. Repetition	pan	panpan
13. Imagistic manipulation (180 degrees rotation of letters)	swimshow	smiwshom

From Kamaljit Singh[KS 85]:

The use of longer passwords and a good condenser to alleviate the delay to encrypt the password.

From National Security Center[NC 85]

The use of pass-phrases such as: The Grass is Green, password: *TGrssGrn*.

However, what about those passwords that cannot be used because of fear that a "cracker" {someone who would dial up your computer and guess at passwords until he/she is successful in entering your account [RB B7]} might guess them? Are those fears valid? Can someone guess your password? One answer to these questions is that human reasoning has been taken for granted in the written guidelines as common-sense issues. Since they are just common sense, some of the suggestions given have never been implemented and have been completely ignored.

I have investigated the validity of eliminating a certain class of passwords from a secure system, based only on common sense (i.e., because someone knows how to guess them). In doing so, I have collected papers and have interviewed computer wizards that have, in one way or another, discovered someone's password (see Appendix A for the method used in the interviews). I also have collected papers showing what trade-offs are used in different password-selection mechanisms.

Some of the people that have guessed passwords have done so not only for fun or malice, but in their roles as system managers or security personnel in checking the security of their systems. Their premise is, "If I cannot break it, then at least I know I am secure from someone up to my level of expertise."

Users, by nature, will look for user-friendly passwords, but because a user-friendly password may be extremely vulnerable to cracking, system managers will probably want to use a random password-generation algorithm that is more robust (i.e., will produce a password that is difficult to crack).

By collecting samples of crackers' behavior and trains of thought, I have formulated a model for evaluating whether a password is breakable or unbreakable. In the discussion that follows, I describe the results of my study and present some of the user-friendly password-selection methodologies recommended by the literature.

Password Selection

Many people think that a pronounceable, randomly generated password is the best solution to the problems of crackers, user recall, and security [SK 83]. However, a study by Winitz, Herriman, and Belleross [WB 75] shows that pronounceable words that do not make sense are not remembered as well as ones that have meaning. Cognitive science [JA 85] has shown that, to improve memorization, one must go through an "elaboration" process. This process is the act of associating a word to a meaning or a picture and repeating this association process as much as needed to memorize the word. The more elaboration done while memorizing, the more activation is involved and the more reliable is the recall of the memorized information. Because memorizing *meaningful* words includes both the elaboration and activation processes, it makes the words less tedious to memorize. Therefore, users tend to write down pronounceable generated passwords and thereby compromise the security of the system.

What about pass phrases, password transformation, and so forth (see Fig. 1 for examples) [BB B4, NC 85, WK 87]? All have something in common: if they mean something to the person using them, they will be recalled. It should be emphasized that a strongly associated password will reduce the user's tendency to write it down. Writing down the password is the most common failing of a new user and greatly reduces the security of a system.

Barton and Barton [BB 84] provide us approaches to password selection based on "semantic memory," "episodic memory," or information prompted by the environment. Semantic memory refers to the generation of a password that is a character string associated, for example, with a song or city. I Left My Heart in San Francisco would yield the password **ILMHISF**. The use of episodic memory would generate a password based on some personal experience. Environmentally prompted passwords would be local street names, parks, attractions, or other places or things that have meaning to the user.

Although these three password selection paradigms facilitate user recall, their use without other restrictions may enable unauthorized users to take advantage of widely known information as cues. In fact, from my interviews, I have found these concerns to be valid.

Case Studies

Now, what information can be used to generate an easy-to-recall password? First, a user's name is certainly easily recallable. The user would not need to write it down to remember it. However, a cracker would try something that simple right away.

From all the interviews, it turns out that *most* account names are related to the user's name (either the person or the organization). This enables co-workers to easily recognize the owner of the account. Using this information, crackers tried user names (full names, initials, nicknames, names backward, or capitalized) and found accounts they could break into. One interesting case described in an interview was that of a school's account for its department of economics and commerce, known as ECON/COM for short. The department had a supervisory account with the account name ECON. The password was COM (trivial!).

The same problem is encountered when users employ episodic memory. An example is the case of a user who had a new baby boy [BB 84]. He used the baby's name as a password for a short period of time. During that period, someone guessed the baby's name as the account password, and of course, gained access.

As an example of a password prompted by the environment [BB 84], a college student, looking for more file storage capacity, tried the names of the streets in the college area and the brand names of cars and motorcycles, and gained access to three accounts.

Since the password can be from one to eight characters long on a UNIX system, some users operate on the principle, "Let's memorize as few letters as we can." The reasoning is, "Since I haven't found a good meaningful password and a quick one to remember, let me just memorize a small set of strings." We know by cognitive science that our ability to retain small chunks (5 ± 2) [JA 85] in memory is good, so why not use it?

In the early days of UNIX, users could select a password that was less than three characters long. It also happened that the encryption algorithm used would take a different time on login accounts depending on the validation of the account name. In addition, the encryption mechanism used could be executed from the library a lot faster than through the login process. Therefore, by using a copy of the password file and the encryption call, Morris and Thompson [MT 79] found 551 passwords with less than 4 characters in a collection of 3289 passwords, and every single one of my interviewees found them too. It should be noted too that Morris and Thompson took less than 22 seconds on a PDP-ll/70 to hit a password of this class.

Usually, the password file includes a comment field, which contains pertinent personal information. This is done to identify the owner of the account, where to find that person, and how to contact him/her in case of an emergency. With this information at hand, crackers have hit passwords with the user's telephone number.

While doing a study in New Jersey, Morris and Thompson [MT 79] ran a successful search for passwords using the license plates issued in that area.

It looks impossible for an user to find a password that satisfies Barton and Barton criteria of user-friendly password selection [BB 84]; however, they listed additional requirements that users usually do not think about.

Findings

All the interviewees agreed that the user selection process I have described is breakable since they had been able to get into accounts using those types of passwords. From what I collected, it seems that a password that consists only of letters or numbers is not a secure password because they have successfully been attacked. Morris and Thompson [MT 79] give the following information on how much time it will take a PDP-11/70 to hit a password of just letters:

Number of characters	Time required
1	30 msec
2	800 msec
3	22 sec
4	10 min
5	4 hr
6	107 hr

On a faster machine, such as a 68000-based architecture, the above times would be faster by at least a factor of 10.

So, what are we to do? From my collection of data from literature and interviews, I have found a decided preference to add other information to a password string, such as some numbers, control characters, or any other characters found on the keyboard. (One should become aware, too, that if the method of selecting passwords is restricted to a number in a particular position, the cracker community will probably know about it since more than one person will know the methodology of selection.)

In an empirical study I did last year, I discovered that the more closely associated the password is to the account name, the easier it is to guess the password. I also found that if the password has a highly unrelated meaning and only the user understands how it was formed, no one else could guess the password in 20 tries.

Therefore, from the sources indicated, here are the recommendations I have found for generating a password with a low probability of being guessed:

- [BB 84] Produce passwords closely related to a childhood verse, using numeric representations for certain words and homophones for others. For example: **14MUNNY, 24SHOW, 3GTREDDY, 42GO.**
- Produce passwords by using the shift key. For example, by using the shift key, 6/6/1944 will be transformed to **¢?¢?!($$.**
- [WK 87] Select two unrelated passwords and at least one number or control character, combine the two words and the control character or number together, and make it fit the length permitted for that system to use. For example, the unrelated words dog and butter and the number 4 yield **dog4butt**, a password of eight characters, the maximum length for UNIX systems.
- [LB 87] Select a word or phrase as a key to relate numerical values to letters. For example, using "black night," let b be 0 (zero), the letter l (ell) be 1 (one), and so on. Thus, a numerical password such as 385 is represented by the letters **chn**. This way, the numerical password could be posted, without compromise, as long as the number-encrypting phrase is protected.
- Program the system to formulate passwords that work only if responded to by associated passwords. Example: the system says **murphy**, and user answers **law**.
- [Interviewees] Half of those interviewed recommended a password that contains a shift, control, or "magic" character.
- Put a control-control sequence and a control-P in the password (e.g., **a^^bP̂c**).
- Use one capital letter, two punctuation marks, three lower-case characters, one control character, and one space (e.g., **I^Plov ?!**).
- From an excerpt from a selected nontrivial phrase, produce an acronym for the password. This gives you a cue to recall the password by remembering the whole phrase. For example, from a short story (*Alice Sheldon*, by James Tiptree, Jr.): "And I awoke and found me here on the cold hill side." The password is: **mhotchs.**

As we can see, there is a commonality between the articles, journals, and the experts on guessing passwords. We know, then, that we can select a generation scheme that will produce fairly secure passwords. In other words, we can say that at least 70% of the passwords generated through these recommended schemes can survive brute-force attacks.

Ways to Crack a Password

From all the interviewees and from the papers collected, the following checklist shows all the ways in which a person will try to crack a password:

- Pretend to be a legitimate user and ask the operator or system manager to give you a new password.

Jerry Schneider, whose job is protecting data processing, demonstrated that he could steal valuable information simply by convincing data-processing people to cooperate with him. He turned on his time-sharing system and dialed the telephone number for a service he was using, and then incorrectly typed his own password twice. He then called the service by voice on the telephone. Schneider told the operator that he had forgotten his password and asked the operator to please supply it to him. The operator refused, stating that he was not allowed to provide passwords over the telephone. He told the operator he would not be able to continue the service unless he got his work accomplished that night. After calling Schneider back, using the telephone number that appears on the customer list, the operator gave him a new password. The operator did not, at any time, however, verify that he was talking to the legitimate customer named Jerry Schneider.

- Try to break the system through the standard accounts with their default passwords. These are accounts used by each computer company to help install and maintain their systems. Some standard accounts are: **field** under ULTRIX; **SYSTEM, SYSMAN, SALESIN,** and **CUSTMR** under IBM; **guest, user, and anonymous** under UNIX, and accounts installed by packages, such as databases and printers. Nine out of ten of the papers I read and the people I interviewed said this method should be tried first since these accounts have high privileges.

Just recently, while going through my regular walk-through procedure to verify the security of a system, I got into the system by using the password of one of the standard high-privilege accounts the system comes with. A previous reading of the system installation manual gave me enough insight to know what account name to look for.

- Try the system's on-line dictionary and compare any encrypted words with the password field. All the interviewees have done this, gaining access at least 60% of the time.
- Try to look for passwords written down in a temporarily unoccupied office or observe the user typing in his/her password. On a teletype system, it used to be that users would type their passwords before the prompt appeared and would leave the paper right there. In a computer-room environment, trash cans are filled often, and frequent janitorial service is not unusual. Therefore, crackers could go through the cans and take computer paper home to study.

In such a case [RJ 85], a bank cashier watched users of automatic teller machines (ATMs) entering their personal identification numbers (PINs) and recorded them, together with the matching account numbers. The cashier then made dummy cards with valid data on the magnetic stripe and used them with the PINs to obtain cash from the ATMs.

- Try to collect information about users, such as their full names, spouses' or children's names, pictures in their offices, or books that are related to their hobbies. All the interviewees would use this technique if necessary.
- Capture the password file and use the encryption mechanism of that particular system at your own leisure.to check for one- to three-character-long strings and minor perturbations such as the first character of each word in upper case letters. Recently, I found 10% of such passwords on a security-conscious computer system.
- Try users' phone numbers, social security numbers, or room numbers.
- Try the license plates of that area or state [MT 79].
- Tap the line. Use a device like the Datascope (line monitor) that can be quickly connected anywhere on the line between a remote user and the host system. All signals, including passwords, can be read in clear text. With this device, a collection of passwords can be gathered in a matter of hours.
- Use "Trojan horses." Programs that simulate the login process can capture and record the password associated with an account.

A low-privilege user produced a game program and invited the operator to use it in his free time [HW 85]. The operator accessed this game through the system's console, which had the highest privilege level. The user had written a small routine in the games program which, when called from the console, dumped

Table 5
Passwords that have been guessed by crackers.

Default account names	**COM, SYSMAN, CUSTMR, field, guest**
Male names (related to user account names)	**Joe, Cesar, ...**
Female names	**martha, sally, sue, ...**
Dictionary words	**foxylady, password, obscurity, lovers, passwd, friend, random, lover, obvious, rosebud, 12345a, love, arpa, ...**
Profanity	Curses, epithets, or expletives
Car brands	**Camaro, rx7, ...**
Repeating initials	**erggre, rmsrms, ...**
Passwords up to 3 characters long	**a, yes, rms, ...**
Environment	Street names, license plates, campus buildings, room numbers, ...
Repeated characters	**aaaaaa, xxxxxx, ...**

all privileged passwords into the user's file. The passwords were then used to access protected information.

- Access the system through backups.

After entering a UNIX Version 6 system by penetrating the root account, a cracker added a "trap door" to the login source code through which he could re-enter later.

Since the system was used by college students and their accounts were removed at the end of each semester, the cracker had to find a way to keep the trap door in the system. The first problem encountered was the system managerial procedure, which removed student accounts by restoring the system from the operating system's distribution tapes. In this way, the system manager did not have to inspect every directory for corrupted files or executable programs that students might have been using to increase their quota on the system. Although it sounds like a secure approach, in this case the cracker used it to determine where it would be safe to put the corrupted login program for later use.

The cracker acquired the key to the central computer room and access to the tapes and, in less than five hours, saved the corrupted login on the tape, leaving the system running as it was before.

Why did the cracker go to so much trouble? The cracker's purpose was to get more CPU time on the system, not to get to user's accounts and corrupt them. In this way, the cracker could keep the higher privilege to see anything on the system and to acquire more CPU time and file-storage capacity.

This checklist contains a few loopholes that will allow a cracker to gain access, even if a user is using a very good password. However, the tapping of the line, trojan horses, and access through backups can be detected by other mechanisms used by the system manager to provide a secure system.

But what to do about the people who bypass the system and pretend to be legitimate users? I think this problem can be solved by educating the operators on security issues. I would like to emphasize that 64% of the problems described by the interviewees lie in the user's hands. The user who should be responsible for that 64%, but the system manager should help the user to fulfill that responsibility.

Table 5 is a list of passwords that have been used (and guessed) in different systems.

Conclusion

I have shown that some passwords can be guessed. Those passwords are breakable not only because someone can guess or deduce them but also because security handbooks actually tell the cracker what things to look for. This means crackers will try those passwords first before going on to the next step. Users should eliminate those passwords that are known to be breakable.

Below are shown some rules for protecting data and generating more secure and user-friendly passwords. Although the system managers' fear of the guessable password is valid, users should consider this list of recommended rules for developing passwords that are less likely to be guessed.

- Study the techniques that crackers are known to use in learning passwords, especially the ones related to user responsibility, and help the user to defend against them.
- Motivate users to protect and not share passwords.
- Monitor unsuccessful attempts to illegally enter accounts.
- Protect records of password(s).
- Do not publicize, other than in the group, how passwords are selected; i.e., giving out too much information compromises security.
- Do not write passwords down.
- Frequently change passwords, particularly any time a compromise is suspected or any time the password is hard to remember.
- Make sure that passwords are not visible.
- Check password selection under these areas:
  - ✓Does the password satisfy the criteria of recommended passwords?
  - ✓Is the password easy to remember?
  - ✓Is it sufficiently long to ensure a degree of protection?
  - ✓Is it designed for *slow* encryption—i.e., designed to ensure a long guessing process?

There are some disadvantages with this model:

- It would spend computer processing time that, in some communities, cannot be spared.
- The model would help the system manager in performing his/her job more efficiently, but human intervention is still needed since deductive reasoning is required.
- It needs cooperation from the users to provide security. In other words, the users must cooperate by applying rules for password selection.

Future Experiment to Pursue

In addition to the data collected from the interviewees and publications, I would like to propose an experiment in which computer wizards who have already used passwords to break into a system would attempt to determine a password. Each subject would record, in real time, his or her thought processes while attempting to find the password. The subjects would be asked to verbally describe their thinking and to record all the passwords attempted at each stage of their search. The recording of passwords generated by the subjects would further identify the areas of their individual thinking processes and also identify those areas that might be common to all potential crackers in the process of guessing a password.

The experiment should involve at least four groups. Two would have to guess the same password but would not have access to the same amount of information related to the password. In other words, cues would be given to one group to help them guess and no cues to the other. The password to be guessed by the other two groups would require more application of the "activation process" to guess it, thus providing a way to see if any differences develop as a result of the amount of activation process involved.

The testing period would be divided into blocks of 20 tries with a maximum of 4 blocks. The opportunity of being able to try 80 times to guess a password should help us measure the maximum and minimum number of tries necessary to guess a password.

Acknowledgment

I thank the hackers who helped me collect information on their tricks, Mr. Russell Brand for his valuable experiences and knowledge in the area of computer security, and my co-workers and Mr. John L. Thomas for their support.

References

[AS 79] Adi Shamir, "How to Share a Secret", *Communications of the ACM,* Vol. 22,No. 11, Pg 612-613, November 1979.

[BB 84] Ben F. Barton and Marthalee S. Barton, "User-Friendly Password Methods for Computer-Mediated Information Systems", *Computers and Security,* Vol. 3, Pg 186-195, Elsevier Science Publishers, B.V, North-Holland, 1984.

[DP 76] Donn B. Parker, "Jerry Schneider: Ex-Computer Criminal", *Crime by Computer,* Charles Scribner's Sons, Pg 59-70, 1976.

[FD 71] Paul W. Fox and Peter R. Dahl, "Aided Retrieval of Previously Unrecalled Information", *JEP 1971, Pg.349-353*

[HH 84] Harold Joseph Highland, "Passwords and Operating Systems", *Protecting Your Microcomputer System,* John Wiley & Sons, Inc. ,Ch 9. , 1984.

[HW 85] Harold Weiss, editor. "How Passwords are Cracked", *The EDP Audit Control and Security Newsletter,* Vol 3, No. 3, September 1985.

[JA 85] John R. Anderson, *Cognitive Psychology and Its Implications,* W.H. Freeman and Co., Ch. 6 and 7, 1985.

[JC 84] James Arlin Cooper, "Protective Procedure-Based Techniques", *Computer Security Technology,* Lexington Books, Pg 37-47,107, 1984.

[KM 79] Leonard I. Krauss and Aileen MacGahan. "Computer Fraud and ConterMeasures", Prentice-Hall Inc., Englewood Cliffs, N.J. 07632,1979.

[KS 85] Kamaljit Singh, "On Improvements to Password Security", *Operating Systems Review (ACM),* Vol.19, No.1, Pg 53-60, Jan 1985.

[KW 84] Stephen G. Kochan and Patrick H. Wood. "Exploring the UNIX System", Hayden Book Co., Chapters 1-2, N.J. 1984.

[LB 87] Lee Berton, "Low-Tech Steps Can Help Guard Computer Data", The Wall Street Journal, December 28,1987.

[MT 79] Robert Morris and Ken Thompson, "Password Security: A Case History", *Communications of the ACM*, Vol 22,No. 11, Pg 594-597, November 1979.

[NC 85] National Computer Security Center. Department of Defense Password Management Guideline, CSC-STD-002-85, Library No.S-226,994.

[NN 88] NetNews from UNIX wizard. Vols.

[RB 84] R. Leonard Brown. "Computer System Access Control Using Passwords", *Computer Security: A Global Challenge*, Elsevier Science Publisher B.V., North Holland, pgs 129-142. 1984.

[RB 87] Russell L. Brand. "Compcon Tutorial on Computer Security", *IEE COMPCON '87 Conference*, also appeared as UCRL-95908.

[RJ 85] Richard J. Johnstone, "Combatting Fraudulent Attacks on EFTPOS Systems", *System Security: the technical challenge*, Proceedings of the International Conference, October 1985.

[SK 83] Stan Kurzban, "Dozen Gross 'MYTHCONCEPTIONS' about Information Processing Security", *IFIP/Sec'83, Proceedings of the First Security Conference*, North-Holland Publishing Co., Pg 1-21, 1983.

[WB 75] H. Winitz, E. Herriman, and B. Belleross, "Long-Term Recall of Speech Sounds as a Function of Pronounceability", *Language and Speech*, Vol. 18, Pg 74-82, 1975.

[WK 87] Patrick H. Wood and Stephen G. Kochan. "UNIX System Security", Hayden Book Co., Chapter 3, Indiana. 1987.

Experiences with Kerberos

Steven J. Lunt

Bellcore
RRC 1L-213
444 Hoes Lane
Piscataway, NJ 08854
(201) 699-4244
lunt@ctt.bellcore.com

1. Introduction

Kerberos™[1] [2] is a trusted third-party authentication system developed and distributed without charge by MIT's Project Athena.† It is a system being considered by standards bodies and major vendors as a basis for building secure network services.

This paper begins with a short overview of the architecture and features of Kerberos, and explains the uses of Kerberos. The author then relates his experiences in bringing Kerberos up on a private lab network, and gives results of his work to date. The paper also includes a summary of other organizations' experiences with Kerberos obtained from feedback received from a survey posted to the Kerberos mailing list. The paper will conclude with some ideas for future Kerberos work, including work being done at MIT, Bellcore, and other organizations.

This paper is meant to share Bellcore's experiences with others who intend to use Kerberos in the future. It is intended that this paper, given at the Second UNIX® Security Workshop, will generate discussion of experiences by others in attendance who have already incorporated Kerberos into their environments.

2. Kerberos Architecture and Features

Kerberos provides a mechanism whereby clients and servers in a network can communicate in a more secure fashion. Central to the Kerberos model is the key distribution center (KDC), a trusted machine that holds the secret keys for each of the entities it serves. A client using Kerberos may reliably prove it's identity to a server over an open (untrusted) network by passing a *ticket* obtained from the KDC. By passing the ticket, the client and server come to share a common *session key* which they may use to ensure either the integrity or secrecy of their communications.

† It is the policy of Bellcore to avoid any statements of comparative analysis or evaluation of products or vendors. Any mention of products or vendors in this paper is done where necessary for the sake of scientific accuracy and precision, or for background information to a point of technology analysis, or to provide an example of a technology for illustrative purposes, and should not be construed as either positive or negative commentary on that product or that vendor. Neither the inclusion of a product or a vendor in this paper, nor the omission of a product or a vendor, should be interpreted as indicating a position or opinion of that product or vendor on the part of the author or of Bellcore.

2.1 Protocol

There are four steps for a client C to authenticate to a server S. These steps are outlined in Figure 3 and are described in more detail in the following list. This presentation is very terse, and neither motivates nor justifies the protocol. For more detail, see the references.

The following assumes a client C wishes to authenticate to a server S.

1. The client C requests a ticket $T_{c,s}$ from the Kerberos authentication server KDC for the server S.

$$T_{c,s} = \{s, c, addr, timestamp, life, K_{c,s}\}K_s$$

Figure 1. A *Kerberos* Ticket

2. The reply from Kerberos is encrypted in the key of the client, and contains the ticket and the session key. The ticket also contains the session key, and is itself encrypted in the key of the server.

3. The client presents the ticket to the server as proof of its identity. It also passes an *authenticator* as proof of its knowledge of the session key.

$$A_{c,s} = \{c, addr, timestamp\}K_{c,s}$$

Figure 2. A *Kerberos* Authenticator

4. The client may request that the service present proof of its identity (mutual authentication) by requesting a reply encrypted in the session key.

Step	From→To		Message
1.	C→KDC	:	c,s
2.	KDC→C	:	$\{K_{c,s}, T_{c,s}\}K_c$
3.	C→S	:	$T_{c,s}, A_{c,s}$
4.	S→C	:	$\{timestamp+1\}K_{c,s}$

Figure 3. *Kerberos* Protocol Synopsis

The two entities may now either continue to communicate in the clear, or use the session key to encrypt the entire session, or use the session key to create encrypted checksums to provide message integrity.

Users are initially authenticated with a password. The client's secret key K_c is actually a one way function of the user's password. An extra step was added to the protocol to loosen the requirement that the user type the password at every service request. A *ticket granting ticket* (TGT) is fetched with K_c initially so that further ticket requests may be made using the TGT. Thus, the actual protocol replaces step 2 above with:

2a. KDC→C : $\{K_{c,tgs}, T_{c,tgs}\}K_c$

and further requests for service tickets are made as follows:

2b. C→TGS : $s, T_{c,tgs}, A_{c,tgs}$

2c. TGS→C : $\{K_{c,s}, T_{c,s}\}K_{c,tgs}$

2.2 Kerberos Parts

Kerberos is composed of library routines usable for implementing secure network services. It comes with Kerberized versions of *rlogin*, *rsh*, and *rcp*. The Kerberos software also comes with the authentication server code, and code for manipulating the database both locally and remotely. Manual pages and installation guides are included. Below is a list of its complete contents.

- User commands (clients):

`sample_client`	for testing

`rlogin, rsh, rcp,`
`login.krb, tftp`
	Kerberized Berkeley commands

`kinit`	get a Kerberos TGT
`klist`	list your Kerberos tickets
`kdestroy`	destroy your Kerberos tickets
`ksu`	same as `su`, but do Kerberos authentication for root
`ksrvtgt`	same as `kinit`, but get key from file
`kadmin`	remote Kerberos database administration
`kpasswd`	change Kerberos password (special case of kadmin)
`ksrvutil`	manipulate **srvtab**

- Admin commands (used on Kerberos server machine):

`kdb_init`	initializes the master database
`kdb_edit`	add or change entries in the Kerberos database
`kdb_util`	dump or load the Kerberos database to/from an ascii file
`kdb_destroy`	delete the Kerberos database
`ext_srvtab`	create a host-specific **srvtab** from the Kerberos database
`kstash`	stash the master password on the KDC

- Daemons (servers):

`sample_server`	for testing
`kerberos`	services read-only requests for Kerberos tickets from KDC
`kadmind`	services read-write requests to the Kerberos master database
`klogind`	Kerberized `rlogind`
`kshd`	Kerberized `rshd`
`tftpd`	Kerberized `tftpd`
`kprop, kpropd`	KDC distribution client, server
`knetd`	`inetd` for Kerberos services (optional)

- Libraries:

`libacl.a`	access control list libraries
`libdes.a`	DES encryption libraries (not for export)
`libkadm.a`	administration server libraries
`libkdb.a`	Kerberos database libraries
`libkrb.a`	Kerberos application libraries
`libknet.a`	`knetd` libraries

3. Utility of Kerberos

Installing Kerberos on your system will make *rlogin*, *rsh*, and *rcp* more secure in that they will perform stronger authentication and thus are less easily spoofed. Kerberos is also useful in centrally administering passwords used by *login*. It will permit you to use other services which use Kerberos authentication, such as the *discuss* system.

It will not make your system completely secure. Traditional measures in securing your system are still necessary, such as ensuring that passwords are not trivial (since password authentication may still be allowed via *login*, *telnet*, or *rlogin*).

The true utility in Kerberos is as a tool for developers to create secure network services. For instance, Kerberos could be employed to create a more secure version of *ftp*. It has potential as a de facto standard for private key authentication, so its ubiquity would facilitate the interoperability and portability of services which do strong authentication.

4. Installing and Using Kerberos

Kerberos was developed at MIT to run on DEC VAX™ and MIPS, and IBM RT™ PCs running 4.3. It has been reported to be running on Sun3™, Sun4™, and HP® 300s and 800s. I was able to port it to the SPARCstation™, after making fixes which I posted to the Kerberos mailing list. I was unable to port it to the Sun386i™.

There is some initial effort in initializing the Kerberos passwords and creating the various control files, but once Kerberos is operational, there is little administration effort necessary.

The same can be said for users of Kerberos services. They need to initially create a **.klogin** file, initialize their Kerberos password, and learn a handful of simple commands to manipulate their ticket file. In addition, they may need to type their password once or twice a day extra.

4.1 Interoperability With Non-Kerberos Systems

The Kerberized Berkeley *r* commands will first attempt to do Kerberos authentication. If this fails, then the old-style method is attempted. The associated daemons may be installed to either allow or deny old-style *.rhosts* or password authenticated requests. The latter case requires explicit deinstallation of the old *rlogin* and *rcmd* daemons from **/etc/inetd.conf**. The implication of leaving the old daemons running is that those systems will be susceptible to their vulnerabilities. The implication of removing the old daemons is that you will not be able to interact with non-Kerberized hosts.

5. Kerberos' Acceptance

Kerberos *seems* to be well established in the UNIX community, but is Kerberos really being *run* everywhere? We began to suspect that most sites printed the documentation, ran into problems with compilation or installation, and went no further.

5.1 Survey

In March, 1990, a survey was posted to the Kerberos mailing list requesting information on who has Kerberos running, which systems are being used, how many realms are in use, or any reasons why Kerberos is not being used.

We received 21 responses, 10 from universities and 11 from corporations. Aside from MIT, the largest reported installation was 75 hosts. There are several organizations trying to integrate Kerberos into

DEC and VAX are trademarks of Digital Equipment Corporation.
Sun3, Sun4, Sun386i, and SPARCstation are trademarks of Sun Microsystems, Inc.
IBM and RT PC are trademarks of International Business Machines Corporation.
HP is a registered trademark of Hewlett Packard, Inc.

existing services. Below are synopses of the responses.

- MIT has 2 major realms, 3 or 4 others here and there. Athena has about 1500 machines. Mostly DEC (VAX and MIPS), IBM (RT PCs running 4.3). Using rlogin/rsh/rcp, NFS™, AFS™, mail (POP), conferencing (discuss), real-time messaging (zephyr).
- University has several types of machines (VAXen, Sun3) running the Athena software. They have one realm in a master/slave setup. They've used *discuss* over the Internet with MIT.
- Univerity has 1 realm, about 5 hosts (Sun3s) running the standard services. Plan to go to 300 machines.
- University has 3 hosts (IBM RT PC, Sun3, Sun4) under 1 realm, experimentally.
- University is using Kerberos as an authentication service for the VISA protocol developed by Deborah Estrin et al. The plans are to have two realms each consisting of three Sun 3/60s™.
- University is looking into using Kerberos widely, if they can interact with non-Kerberos hosts, and port it easily.
- University would like to use Kerberos and Hesoid to help replace Sun's Yellow Pages.
- University cannot use Kerberos until they find a version of bind that understands Hesiod records.
- Two English universities and a Canadian firm ran into export restrictions for DES code.
- European university had space and performance problems in integrating Kerberos into their distributed application.
- Bellcore's Computer Security District has 6 SPARCstations running SunOS™ 4.0.3 and 4.1 under one realm running rlogin, rsh, and rcp.
- Corporation has one realm running 75 hosts, mostly Suns (SunOS 4.0.3 and 3.5) and VAXen. Some experience porting. Most bugs in application code (e.g., rlogin) rather than libraries.
- Major corporation has one realm running 30-40 HP 300s and 800s. Working on Apollo machines. Using standard Kerberos services, plus a few homegrown ones. Also made some other local mods.
- Corporation has 14 hosts under 2 realms working experimentally.
- Corporation might look to Kerberos for better authentication for their network software product. They have a package to help administration of network services, and would like to apply this to Kerberos itself. Their hardware is Sun3, Sun4, Sun386i, DECstations™, and MIPS boxes.
- Tough bugs, poor software engineering prevented further use. Lack of formal proof of protocol yields low faith in security. Evidence from Oakland conference suggests formal methods not sufficient for assurance.
- Major corporation would like *login* to not force a particular policy.
- Corporation plans on trying it in the future.

AFS is a trademark of Transarc Corporation.
DECstation is a trademark of Digital Equipment Corporation.
SunOS, Sun 3/60, and NFS are trademarks of Sun Microsystems, Inc.

- Corporation plans to evaluate Kerberos.

In addition to those who responded to our survey, there have been many contributors to the Kerberos mailing list from both universities and corporations.

5.2 Kerberos in the Industry

Kerberos will be the authentication mechanism for the Open Software Foundation's Distributed Computing Environment (OSF^{TM} DCE). Kerberos will also be in the next release of ULTRIX^{TM} and Berkeley UNIX Release 4.3.

5.3 Kerberos Version 5

This paper discusses Version 4 of Kerberos, the current implementation. Currently, the architects of Kerberos are drafting Version 5 of Kerberos. The Version 5 RFC draft document[3] is available via *ftp*. Major changes in Version 5 include new ticket options such as forwardable, proxy, postdated, and renewable, a two-TGT user-to-user authentication mechanism, and machine address type independence.

6. Open Issues

Some of the following issues are being addressed in Version 5.

6.1 Passwords

Kerberos uses traditional passwords for authenticating users. Thus, all of the vulnerabilities from the use of passwords still remain. *kadmind* should do a triviality check on all key (password) change requests. Kerberos should facilitate the use of smart cards or challenge-response cards.

6.2 Timestamps

Kerberos uses timestamps to prevent the reply of messages and to ensure the timeliness of credentials (so that they are not good indefinitely). Since the Kerberos authentication server sets all timestamps, each host in a realm must have their clocks loosely synchronized. This is often too strong a requirement in a large network of hosts.

6.3 Portability

Although care was taken to make the encryption routines portable across architectures, less care was taken in the code which packages and unpackages data across the network. Kerberos Version 5 will use ASN.1 encoding of data for portability and standardization.

6.4 Dependence on Lower Protocol Layers

Kerberos is written to utilize the Berkeley *socket* interface. It also has as part of the ticket the IP address of the client. Kerberos should be usable over any transport level interface, and should be portable to System V platforms which have no *socket* interface.

6.5 Administration

There should be a more automated way of installing the various Kerberos services, given the install options that are available (e.g., *rlogin*).

OSF is a trademark of the Open Software Foundation, Inc.
ULTRIX is a trademark of Digital Equipment Corporation.

7. For More Information

There are several ways in which to learn more about Kerberos. First is to grab the source and documentation with *ftp* and read the documentation. Kerberos is available via anonymous *ftp* from ATHENA-DIST.MIT.EDU (18.71.0.38, cd to **pub/kerberos**). The papers in the references are also available via *ftp*. Next is to read the Kerberos Usenet newsgroup `comp.protocols.kerberos`. There is also an electronic Kerberos mailing list which can be subscribed to by mailing a message to `kerberos-request@athena.mit.edu`. The mailing list messages are also sent to the `comp.protocols.kerberos` newsgroup.

8. Summary

We found Kerberos to address client-to-server authentication in a greatly needed way. However, we found it to not address user-to-host (i.e., login) authentication more than existing methods, aside from facilitating centrally administered passwords. We also found Kerberos to port to only a limited set of architectures and environments. Nevertheless, we feel that Kerberos has potential for use throughout the community, especially if these problems are better addressed.

REFERENCES

1. Miller, S. P., Neuman, B. C., Schiller, J. I., and Saltzer, J. H., *Section E.2.1: Kerberos Authentication and Authorization System*, M.I.T. Project Athena, Cambridge, MA (December 1987).
2. Steiner, J. G., Neuman, C., and Schiller, J. I., ''Kerberos: An Authentication Service for Open System Networks'', *Usenix Conference Proceedings, Winter 1988*, pp 191-202 (February 1988).
3. Kohl, J., Neuman, B. C., Steiner, J., *Kerberos Version 5 RFC*, Draft 2, M.I.T. Project Athena, Cambridge, MA (November 1989).

Public Key based Authentication using Internet Certificates

Joe Tardo, Kannan Alagappan, Richard Pitkin

Digital Equipment Corporation
550 King Street, LKG1-2/A19
Littleton, MA 01460
sphinx@dsmail.enet.dec.com

Abstract

The Internet community has recommended a key management architecture and infrastructure based on the use of public key certificates for privacy enhanced mail. Our approach is to use these same certificates for authentication of principals and applications, thereby increasing overall network security with essentially the same administrative overhead. In this paper we present Sphinx, a public key based authentication service for open global networks using Internet certificates.

1. Introduction

Currently, the only cryptographic authentication service publicly available is Kerberos, which is symmetric (secret) key based [5]. Secret key technology works well in the local environment. However, public key technology exhibits substantially more favorable scaling and key management properties when applied to large (unreliable) networks with no globally trusted authority. Furthermore, organizations planning to secure both mail and remote login will need to manage a certificate repository for privacy enhanced (PE) mail and a Kerberos realm server for local authentication. In this paper we present a common paradigm for authenticating mail and user applications, thus avoiding the burden of managing both mail certificates and authenticator keys.

Sphinx is a self-contained, portable implementation of a public key authentication service to be used primarily in UNIX[1] environments. It has capabilities comparable to Kerberos V4 with extensions for authentication forwarding (delegation). Authentication forwarding is a mechanism that lets one principal authorize a second principal to act as its representative for a limited time interval; this capability is very useful in many client-server interactions. Public key certificates used by Sphinx comply with X.509 [7], the syntax used for Internet PE-mail certificates [2].

Ideally, principals get Sphinx certificates by name from some ubiquitous naming service. Unfortunately, such an infrastructure is not available. In the interim, Sphinx employs a *Certificate Distribution Center* (CDC), which is analogous to Kerberos's Key Distribution Center (KDC). A CDC keeps a database of principal certificates whereas a KDC is a repository for Kerberos client keys. A major distinction between the two is that a CDC requires less security constraints. For example, a CDC does not need to be trusted and it can be unsecurely replicated for higher availability. If the CDC is compromised, this might result in denial of service for principals, but would not allow a penetrator to subvert the authentication scheme.

Principals first create their long term RSA keys and certificates off-line using *certification authority* routines. Principals then need to place their certificates and encrypted secrets in the CDC (via an enrollment process), which may be handled by an administrator. To reduce the administrative burden, Sphinx has a *Login Enrollment Agent Facility* (LEAF) that allows principals to enroll themselves. Another role of LEAF is to protect encrypted principal secrets from off-line password guessing attacks; consequently, principals who wish to establish their identity locally must access LEAF to obtain their encrypted secrets, which are passed securely to the requested principal. In both cases, LEAF authenticates the principal's identity before providing the requested service. Compromising LEAF would only expose encrypted secrets to password attacks; it would not enable adversaries to subvert the authentication scheme.

[1] UNIX is a registered trademark of AT&T.

Finally, Sphinx provides run time library routines for applications to access authentication services.

2. How does it work ?

Sphinx is based on concepts derived for Digital's Distributed System Security Architecture, described by Gasser, et. al. [1] and Linn [3]. The architecture uses symmetric key cryptography for providing data origin authentication with data integrity. It also uses asymmetric (public) key cryptography for key distribution, certification, and delegation. The preferred algorithm for asymmetric key cryptography is the RSA cryptosystem [6].

With RSA algorithms, "strong" authentication mechanisms can be easily constructed. Each principal creates a public/private key pair. The public key can be published in a name service or any non-secure server, while the private key is kept secret. Cryptographic operations using the private key and public key are inverse functions of each other and can be performed in either order.

A principal that is trying to authenticate its identity is referred to as a *Claimant*. A principal that is trying to verify another principal's identity is referred to as a *Verifier*. A Claimant can authenticate its identity by signing a message with its private key, and sending it to a Verifier. The Verifier would verify the message by using the Claimant's public key and thus know that the message came from the Claimant. If mutual authentication is desired, the Verifier creates an acknowledgment, signs it with the verifier's private key, and the Claimant verifies the mutual response with the Verifier's public key and thus know that the Verifier received its message.

The authentication algorithms used by Sphinx are more complex than the simplified description above, because they are required to deal with a large number of subtle security threats. For example, to prevent private keys from being vulnerable for long periods of time, user workstations generate RSA delegation keys for each login session. Another threat to authentication is replay. Sphinx uses a timestamp-based scheme to handle replay detection.

A basic requirement for authentication is that public keys for remote principals be accessible and valid. In Sphinx, each principal has one or more certification authorities who it is willing to trust to sign/issue certificates, known as *trusted authorities*. Public keys in certificates are obtained from the CDC and verified as having been issued by one of the principal's trusted authorities. This approach allows organizations in widely separate parts of a naming hierarchy to authenticate one another without requiring a common trusted root.

Trust in Sphinx emanates from the set of authorities a principal is willing to believe. Principals trust Certification Authorities to act responsibly and, in particular, to properly authenticate identities before issuing certificates. The certificate distribution mechanism itself is not necessarily trusted.

3. Overview of Sphinx Authentication protocol

With Sphinx, users create *claimant credentials* (which requires that they know a password) in order to cryptographically prove their identity for services. Analogous to Kerberos, claimant credentials consist of a login *ticket* and a delegation or session RSA key. A ticket securely identifies the principal for a ticket's lifetime since it is signed with the principal's private key. Credential structures enable peers to establish security contexts with each other.

Users authenticate to servers by creating an *authentication token* that transfers a DES key encrypted using the target server's public key. This public key is verified to have been issued by one of the user's trusted authorities. Users send an authentication token which (mostly) consists of an *authenticator*, encrypted DES key, and a ticket. An authenticator, which is to be used only once, provides information about this particular authentication instance. In the case of delegation, a token also includes the session private key encrypted using the DES key.

A Server receives the token, extracts the DES key, and verifies the identity of the principal using the user's public key. If the authenticator is within the validity interval and it is not a replay, the identity of the peer is returned to the application. If mutual authentication is requested, the target sends back a mutual response. If delegation is requested, claimant credentials are established on the remote host.

Similar to Kerberos, peers who have authenticated each other can use data integrity and data origin services in per-message exchanges using the shared DES key. Sphinx's service interface is based on the Generic Security Service Application Program Interface (GSS-API) specification [4]. The GSS-API is intended to hide applications from the underlying mechanisms being used for establishing security contexts.

More details of the algorithms can be found in [8].

4. Status

Currently, a preliminary version of Sphinx is undergoing limited testing among a small community of users. A more complete version that includes a number of optimizations is planned for the near future. Early performance figures are encouraging, indicating an initialization time averaging 2-3 seconds on a 13 Mips DECstation 3100. Sphinx has been integrated in the Berkeley r-tools, such as rlogin, rsh and rcp. Authenticating versions of the Berkeley r-tools are in use. We plan to integrate Sphinx with additional applications, such as ftp and telnet.

5. Acknowledgments

This project was strongly influenced by the work of Morrie Gasser, Andy Goldstein, Charlie Kaufman and Butler Lampson. Many thanks go to Shaike Artsy, Charlie Kaufman, John Linn, Andrew Palka, and Percy Tzelnic for their support and review of earlier work.

6. References

[1] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, "The Digital Distributed System Security Architecture", *Proceedings of the 12th National Computer Security Conference*, Baltimore, MD, pp. 305-319, October 1989.

[2] S. Kent and J. Linn, "Privacy Enhancement for Internet Electronic Mail: Part II — Certificate-Based Key Management", RFC 1114.

[3] J. Linn, "Practical Authentication for Distributed Computing", *1990 Security and Privacy Symposium*, Oakland, CA, May 1990.

[4] J. Linn, "Generic Security Services Application Program Interface", UNIX Security Workshop, Portland, OR, August 1990 (to appear).

[5] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer (Massachusetts Institute of Technology), Project Athena Technical Plan Section E.2.1, "Kerberos Authentication and Authorization System", 21 December 1987.

[6] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", CACM, vol. 21, no. 2, pp. 120-126, February 1978.

[7] CCITT Recommendation X.509 (1988), "The Directory Authentication Framework".

[8] J. Tardo and K. Alagappan, "Sphinx: Global Authentication Using Internet Mail Certificates", in preparation, July 1990.

[illegible]

[illegible]

[illegible]

[illegible]

Acknowledgements

[illegible]

References

[1] [illegible]

[2] [illegible]

[3] [illegible]

[4] [illegible]

[5] [illegible]

[6] [illegible]

[7] [illegible]

[8] [illegible]

SYSTEM DESIGN AND VERIFICATION FOR SECURE APPLICATIONS UNDER UNIX

R. B. Neely
Ford Aerospace Corporation

ABSTRACT

This note addresses the problem of modeling trusted processes and systems that contain them, with an underlying secure Unix operating system assumed. It describes an associated problem, sets the stage for a solution, provides the sketch of a solution, and embodies the solution in an example.

1 The Problem

Consider a secure general-purpose operating system that has been demonstrated to meet a general set of security requirements (the system-specific policy). In spite of whatever work has gone into assuring that the security requirements are met (and that is sure to be considerable work, indeed), in a real operating environment, a conflict is sure to arise between usability of the system and maintaining the security assurance. This problem is the result of the following three observations:

a. "Trusted" functionality (i.e., functionality that is outside the constraints of the security policy) beyond that provided by the original OS trusted computing base (TCB) is needed. Example: a multilevel secure (MLS) communications server.

b. It is too expensive to "break the seal" on the delivered TCB by modifying it in any way and so forcing its re-verification.

c. Therefore, the security model proved for the TCB must (by definition) be violated by a "trusted" process, using "hooks" (one could say holes) provided by the original TCB.

As a result, security-critical software may exist that has not been subjected to the level of verification of the original TCB. Even if some attempt is made to verify the trusted processes, the approach is nearly always fragmented, in that no system-wide, unified verification is ever accomplished. Thus, it is never shown that the entire system, including the trusted processes, satisfies an "entire-system-specific" policy. We consider next what kinds of approaches might solve this problem.

2 What a Solution Needs to Do

Trusted applications need, by tacit definition, to do things that violate the security policy that the underlying system (e.g., a secure Unix) is supposed to enforce. Nevertheless, it is presumed that such an application is subject to a set of requirements of its own that it must enforce — otherwise, it might do anything. (What a lovely place to install a Trojan Horse!) To apply this observation, it is necessary to consider the security requirements of the encompassing larger system of which the secure applications, as well as the underlying Unix system, are a part. We must consider what is to be demonstrated, and about what it is to be demonstrated. Each subsystem and component must be demonstrated to meet *its own* security-relevant requirements. As a result, other portions of the system (including the system as a whole) are entitled to assume that it does so. Note that an application that is used in a critical way *is* (by definition) "trusted," but we do not know whether it is trust*worthy* until we have demonstrated something about it! For example, we may be so obtuse as to use a downgrader produced by "Red Star Computing," and if we do so, it is indeed a trusted application. However, its trustworthiness is another matter.

One way to handle secure applications is to generalize the concept to applications having any set of special, critical requirements. This broader view not only solves a much larger set of problems; it also (given our and others' experience) results in a *simpler* approach. One example of a critical requirement that is not directly a traditional security requirement (i.e., not protection of information from compromise) is a requirement that each auditable event be captured in an audit message that is written to an audit log. Another example is a requirement that (1) members of a certain specific class of decisions must be offered to a human mediator, but (2) the mediator must *not* bothered with other, minor, decisions. A further example is a "disaster control" policy, which requires that certain "really bad" things do not happen, such as a file system becoming inconsistent.

3 Sketch of a Solution: A Structured Verification Paradigm

We suggest a solution, which we call a "structured verification paradigm." The first step in this structured way of looking at systems is a structured representation of "security" properties, or policies. The reason I put "security" in quotes, is that I really mean any (sub)set of requirements deemed especially critical by users of the system. With this approach, the policy of a system refers to the behavior of the system as a whole, and might not apply directly to some or all of the system's components. Such a policy (as is normally the view of a set of requirements) is to be levied on the system's external behavior, i.e., with the system viewed as a "black box."

We then look at subsystems and components of the system. As with any requirements analysis, requirements of the system are derived to more and more interior components. Each such component is treated as was the entire system: the derived requirements describe its behavior viewing it as a black box. Subsets of critical requirements are derived to components along with the full set of requirements.

The verification (assurance demonstration) task, then, is demonstrate at each node of the decomposition tree the following paradigm: If those components interior to the component represented by the node under scrutiny satisfy their individual policies, then the node component will satisfy *its* policy. At the lowest level of the decomposition tree, a separate method must be used to demonstrate that each lowest-level component of the actual implemented system indeed satisfies its policy. This is usually termed the "code correspondence analysis," though perhaps "implementation correspondence analysis" would be more appropriate, since some of the TCB components will be in hardware, rather than only software.

4 An Example

Consider an application whose job it is to automatically downgrade information from SECRET to UNCLASSIFIED. The information has presumably come from a system-high system, where the information all had a SECRET classification. Yet the much of the information (in terms of content) is UNCLASSIFIED, and that portion must be transmitted to uncleared users. The downgrading is to be done based on a list of keywords. If any of the keywords appears in the text of a message, than the classification of the message remains at SECRET. Otherwise the message is downgraded.

Let us suppose that the system consists of

- a CPU, memory, and MMU capable of supporting a MLS Unix
- any number of peripheral devices, including two communications lines, one marked SECRET, the other marked UNCLASSIFIED
- a secure (MLS) Unix OS
- any number of "untrusted" processes
- the (trusted, MLS) downgrader process

With one exception, the policy of the Unix OS itself can be a standard "no downward flow" policy, with each process possessing a sensitivity label as a subject, and each file, device, etc., possessing a sensitivity label as an object. The exception is that certain executable files may be marked as `<MLS>`, possessing a range of labels rather than a single label. When such a file is executed, the associated process is marked with the label range of the file. Then information is allowed to flow *to* the process as if it possessed the highest label of the range, and allowed to flow *from* the process as if it possessed the lowest label of the range. Of course, such a system must be administered so that only software that has been properly verified can be used with this MLS status. "Properly verified" means that it has been demonstrated to meet the downgrading policy described above at a level of assurance acceptable for MLS processing of information.

What, then, of the entire system that includes all the hardware and software indicated above? It must have a policy (and be demonstrated to meet such policy) that is a partial amalgamation of the no-downward-flow policy of the Unix OS and the downgrading policy of the downgrader application. It might sound something like "No downward flow of information *unless* the information came from such-and-such source, and contains no keywords from a special list." The architecture of the system must be described (and implemented) in such a way that the entire system can be shown to meet this policy, given that (1) Unix satisfies its own policy, (2) the downgrader satisfies its policy, and (3) the "security-critical" portions of the hardware satisfy their requirements.

5 Conclusion

This example depicts a possible real system that requires more analysis than just making sure that a Unix OS meets generalized security properties, in order to meet the security properties of the entire integrated system. Most of the systems that we develop at Ford Aerospace possess such heterogeneous requirements, some of which are substantially more critical than others. Existing guidelines for developing secure systems offer a starting point (e.g., the *Trusted Computer System Evaluation Criteria*[3]), but real-world systems with complex sets of requirements demand that we go well beyond the starting point, as indicated in Baker[1], Landwehr[2], and Neely[4]. We Unix-oriented folks (as I believe) tend to be creative people, and I am sure there is no limit to the number of effective approaches to providing a high degree of assurance that Unix-based (or other) systems meet particular critical requirements. There have often been confusing messages regarding assurance verification from the computer security community. Perhaps the Unix security segment of that community can provide some clarity that might be (albeit unintentionally) accepted at large.

6 Bibliography

1. Baker, P. C., G. W. Dinolt, J. W. Freeman, M. D. Krenzin, and R. B. Neely, "A1 Assurance for an Internet System: Doing the Job," CSO-TR1134A, *Proceedings, 9th National Computer Security Conference*, Gaithersburg, Maryland, September 1986, pp. 130-137.

2. Landwehr, C., and H. O. Lubbes, "Determining Security Requirements for Complex Systems with the Orange Book," *Proceedings, 8th National Computer Security Conference*, Gaithersburg, Maryland, September 1985, pp. 156-162.

3. National Computer Security Center, *Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, 1985.

4. Neely, R. B., and J. W. Freeman, "Rigorous Integration of Sources of Assurance," *Proceedings, COMPASS '86* (Computer Assurance), Washington, D. C., July 1986.

Security Considerations of Going to a UNIX-Based Supercomputer Operating System

Gary G. Christoph, Ph.D
Worker Systems Section
Computing and Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

ABSTRACT

Cray Research, Inc., is distributing a secure version of their UNIX*-based (AT&T System V) operating system, UNICOS,* for Cray Supercomputers. This operating system is designed with the objective of meeting Orange Book (TCSEC) B2 certification criteria; submission for certification of UNICOS is planned. The security extensions to the operating system are designed to support security for a stand-alone machine interacting with users through front-end computers.

We have several Cray machines, and other computational resources, tied together in our own locally developed network using locally developed protocols; in this environment, a Cray YMP to us is just another resource. Thus the reality of our networked installation does not fit neatly into the security model implemented by CRI in UNICOS. Also, DoD style security requirements do not match our local requirements, which derive primarily from developmental/scientific computational needs.

This talk will focus on the changes we consequently feel important to make to UNICOS to satisfy local practical and cultural security requirements. Practical requirements are satisfying intermachine communication needs over our local multi-classification level heterogeneous network in a secure way (e.g., what to do about *rcp*, e-mail, etc.?). Cultural requirements are changes to UNICOS we feel necessary because our users, being long accustomed to our local Cray timeshared operating system, CTSS, do not correctly predict how UNICOS works, and we have become concerned about inadvertant disclosure (e.g., implicit inheritance of access rights).

Our secure supercomputers and other resources are linked by a high speed packet-switching network, accessible to user terminals through terminal concentrators and distributed processor gateways. Users are authenticated and verified at logon by a network security controller. A user logs onto the network specifying a machine, password, account code and desired security level. Once logged on at a particular security level, the user may, without reauthentication, connect to other machines in the network accessible at that security level. All network messages embody the security level and other critical information in headers that are handled by either the kernel or trusted processes (daemons).

Since it is only possible to log onto the secure network from terminals located inside the physical security areas, and since admission to the physical security areas is rigidly limited to personnel having appropriate clearances, primary concerns we have are: 1) leakage of information between security levels (through either inadvertant granting of access rights or disclosure to users not having suitable need-to-know for that data), and 2) Trojan horse attacks (by means of insufficiently scrutinized imported scripts or programs).

We have chosen to attack the first, a cultural concern, by limiting the *umask* and *chown* commands to trusted (root) processes. We prefer, instead of users sharing programs and data by bestowing access, that a file be directly transferred from one user to the other, subject to stringent requirements and audit trails. The additional control also limits the giver's ability to insert a trojan horse into a user's search path: an explicit action by the recipient is necessary to bring the given file into his directory space and activate it for execution.

Because we are operating Secure UNICOS in a protected environment, some of the CRI protections are not relevant (e.g., */etc/password* is maintained only for user information and is not used for logon validation), or are troublesome (we find it appropriate to disable a user's ability to increase his security level during a session). However, there are also mismatches between UNICOS controls and our network security controls; the latter enforces neither compartments nor ACL's, and these are not easily added to our network protocol. Our direction in using IP is to exploit the security options extensions field and use a network authenticator and a gateway checking mechanism.

* UNIX is a registered Trademark of AT&T; UNICOS is a registered Trademark of Cray Research, Inc.
Los Alamos National Laboratory Publications Reference No. LAUR-90-1790

Perspectives and Solutions
for
Increasing Security in UNIX System Administration:

A Discussion of Security Related Topics
for
Effective Administration of a Large Site.

Bjorn Satdeva

/sys/admin, inc

ABSTRACT

Administration of a large UNIX site is by no means easy since the tools provided by standard UNIX implementations are never quite good enough. The explosive growth in network sizes over the last few years has resulted in larger and more complex sites but, yellow pages notwithstanding, very few new significant tools to assist system administrators in maintaining networked sites. It has therefore become increasingly difficult to keep a site reliable and secure. This paper discusses some simple but effective and proven methods that aid central administration of a large number of machines on a local network. The paper covers automatic file distribution, automatic security audits, and site policies.

1. Introduction

It's unfortunately a fact of life that most system administrators are overloaded in their daily work and have little time for improving site reliability and security. However, in the failure to do so, they are unwillingly making a significant contribution to their own work load. A site where adequate measures for site reliability and security has been integrated into the daily work encounters fewer problems requiring an immediate solution in my experience.

The system administrator can reduce the daily workload through careful planning. However, there are a few select areas where a system administrator at a site prone to many "firefighting" issues can obtain some clear results in a reasonably short time. This paper discusses three such areas.

- The rdist program performs automatic file distribution and keeps specified (often system-) files updated across the network. This can increase reliability and certain aspects of security. This paper is not an rdist tutorial, but does discuss some practical guidelines for utilizing rdist as an effective system administration tool.
- Use of automatic security audits can keep a larger number of systems reasonably secure. Several security audit programs will be discussed, including some examples of what can be uncovered with these tools.
- Establishing of policies is a non-technical issue addressed here. Lack of policies and long range plans for a site, from my experience, always causes problems in day to day operation.

All of these topics are presented from the point of view of a Site System Administrator.

2. Use of rdist.

System administrators who have automated part of their daily routine often find themselves relieved of much of the burden that kept them busy before automation. This is specially true for automatic distribution of system configuration files like hosts, group, and aliases, assuming that the site is not using yellow pages (many people consider YP a security risk).

Automatic file distribution is most easily implemented with the rdist [8] command from 4BSD and now available in many other UNIX implementations. This utility copies files to networked machines by consulting a configuration file of what files should be distributed and to which hosts. However, neither the detailed workings of rdist nor the syntax of the configuration file will be discussed here.

When a site starts using rdist for automatic file distribution, it is important that the system administrator thinks in administrative rather in engineering terms. It is a common mistake to specify all distributed files in one big rdist file and then distribute everything during every night. This is an inefficient means to achieve the administrative goal, as it is desirable to distribute some files immediately after modification (sometimes several times a day), while other files (typically programs) will only need to be distributed occasionally. Being selective in the number of files distributed frequently will minimize both the network traffic caused by rdist and the time it takes to update all hosts.

The first step in the selection process is to examine the files that are candidates for automatic distribution, and decide how often each should be distributed. Some typical examples are:

- Network configuration files which change frequently and which must be kept identical over a large number of hosts. Typical filenames include /etc/hosts, /etc/ethers, and /etc/printcap, These are distributed hourly.
- System configuration files which are locally modified and therefore must be downloaded to new systems at installation time. Such files are typically /etc/rc or /etc/rc.d/* and are candidates for a daily or weekly distribution.
- Files which together make up the body of the local maintained software are also good candidates for periodic distribution. Such files can be found in, e.g., /usr/local or /usr/lbin (depending on local policies) and are candidates for either weekly or monthly distribution. The frequency of distribution of this type of files should be determined by the need to keep exact copies rather than the need for distribution after updates. The system administrator can start a distribution manually if one or more files has been updated.

Seen from the perspective of site security, rdist is an attractive way to synchronize certain files across the site and eliminate the risk of updates to only some of the hosts. However, unless special care is taken, rdist can be a two-edged sword. The host which is doing the rdist ('the network master'), is controlling important files on most, if not all, of the machines on the site. The access to the network master must be restricted as much as practicable in order to protect the site from both well-meaning users and hostile intruders. It is therefore recommendable that the host used for the rdist is used for this purpose only. The most secure result will be obtained, if the network master trusts no other host (empty /.rhost and /etc/host.equiv files) and all network services not required by rdist are turned of. Most importantly, rlogin, telnet, and ftp should not be provided (if at all possible). Of course, the machine must be located in a physically secure area or most of the above protection will be in vain.

It should also be noted that the use of rdist can cause delays between master file update and the arrival of that file on remote machines. While such delays would be unacceptable for some applications, it has been my experience that this type of delay is of no practical consequence for maintaining system configuration files across a network, unless the changes are very radical and severe. In such rare cases, other methods should be applied.

3. Use of security audits

One of the big challenges to a UNIX system administrator is a site's security. This is difficult with only one system and becomes almost impossible when there are hundreds or even thousands of systems. Security can only be achieved through automatic checks which bring security problems to the attention of the system administrator. Tools that perform these checks are called security audits.

Many problems are caused by simple problems which can be fixed easily if the system administrator knows about them. Typical problems involve incorrect file permissions (which can be used to gain unauthorized access or just plain break the system). If the system has an automatic security audit facility which is run regularly (e.g., nightly for most checks), many trivial problems (like wrong access permissions and illegal device and SUID/SGID programs) can easily be corrected.

There are several automatic system audit packages available in the public and semi-public domains. Possible the best known is COPS [1], but there are other packages (like *secure* [7], and SPY [2]). COPS and *secure* are the most interesting in this context, as the sources are publicly available (COPS in the USENET archives, *secure* in the UNIX security book). SPY is a proprietary package from HP, but it has been described in Bruce Spencer's paper "Spy: A UNIX File System Monitor", which contains detailed implementation information, as well as some source code samples. The COPS programs, though, are easiest to install and use without modification. The *secure* package is intended as an educational example and needs some work before it is practical in a real-life situation. However, it is still a very good starting point both for practical implementation of a UNIX security audit and for an education in UNIX system security.

I have been using these types of programs routinely now for several years and am finding them to be very convenient for uncovering the kind of security problems outlined above. In fact, on all sites where I have been using these kind of tools, a number of security violations have always been uncovered in this way. Some examples include:

- At our own site (sysadmin.com) we have been using a modified version of *secure* for a long time. When we installed COPS, and started using that package in addition to *secure*, we discovered that the */usr/admin* directory was publicly writeable. This was of little concern, because we do not use the System V *sysadm* utility (which we consider naive and very incomplete). However, the directory access was still a security violation and we therefore re-installed the entire contents of that directory, after fixing the access permissions.

- At one client site, some of the users had a *su* type program where the user authentication part of the code was removed. It was installed in their private bin directory as "..." (three dot's). The *secure* utility uncovered these programs. We removed the SUID permission bit and notified both the users and their immediate manager.

- At another site, where I installed a number of workstations, a locally developed *secure* look-alike found a large number of (>500) security violations on the virgin installed machines, some of them serious (e.g., world writeable root and /etc directories, as well as a world writeable crontab file).

  In general, I have found that the work of preparing the control files for the *perm* utility (a part of the W&K *secure* utility, which checks file access permissions) is tedious, but rewarding. From the use of this utility, I have learned that most vendors ship their systems with a number of incorrect file ownership or access permissions.

As the above shows, the use of automated audits can be very helpful, specially at a larger site. However, a word of caution is necessary. If a security audit gives a system a clean bill of health, this does not guarantee that there are no security risks on that system; it guarantees that the audit did not find any. A security audit should therefore never lead one to complacency but rather should be though of as a convenient tool which can be of help in finding some of the worst blunders.

4. Establishing of policies.

I know this topic is not very UNIX-like, but the lack of policies is the source of uncounted problems at many sites. In theory, policies are the responsibility of the upper management, but in reallity they are the responsibility of the system administrator/manager. Because upper management at most sites does not understand the issues of system administration, it is the system administrator's responsibility to educate them of what is needed, as well as outline proposals for such policies.

Many of a site's human problems are related to the lack of a policy and procedures on the site. Establishing clear policies known to all users of the site will eliminate most problems, since the established policies provide decision strategy. This will not only aide the system administrator in the daily work but will also help to prohibit random policy decision are made quickly when a specific problems occurs. In most cases, decisions made in the heat of battle overreact to a given situation and result in policy which users, management, and the system administrator alike find uncomfortable in the future.

Below is a discussion of some of the suggested areas where explicit policies are recommended, although the exact needs will depend on the individual site (factors like the number of machines, the number and categories of users, and what kind of services are offered).

4.1 Establish a Site Policy

The Site Policy is the constitution for the site. It must describe in general and non-technical terms the policies that the sysadmin and the users must follow when using the site's equipment and software. Through it, top management states the policies governing the site in general terms and provides the sysadmin with a platform upon which more specific documents can be built. As an example, the site policy should require backups be done and specify a frequency, but should not specify any details of how they are done. The purpose of the policy is not to establish a mindless bureaucracy but rather to create a foundation upon which the Unix Site Manager can build the administration. It is therefore necessary that upper management sign off on the site policy. In doing so, they make a statement of support for the policies and for the system administrator's continued work in this area.

4.2 Security Policy

The security policy is the most interesting document in this context. Security was not taken seriously at many sites before the fall of 1988 when the Morris Worm hit the Internet. Since then, many sites are at least paying lip service to security and at some sites, management has expressed real concern for the security of the site.

The security policy should not only outline traditional requirements, like who is authorized to request new user accounts, how modem line security is implemented, and use of security audits, but should also deal with problem of what to do if an intruder has been detected or a neighbor site calls up and declares they have been intruded from your site, or if the FBI, as in Clifford Stoll's [6] case, asks you to keep a security hole open in order to catch a system intruder.

The security policy must outline how to deal with a situation in which hosts have been compromised or services have been denied, although an overwhelming part of all security problems, in my experience, are caused by the mistakes of well-meaning users or system administrators.

As with any other policy, it is important to use common sense when establishing the security policy. A site should not be made so secure that more is spent to protect a site than the data is worth or that systems are made so secure that it becomes difficult to get real work done.

The above are just some highlights of what should be considered in a security policy. This area is vast and very complex. However, one of the more positive tones in the picture, is that an Internet working group is currently putting together a site security policy handbook which may be able to be used as an aid in building a security policy. This work will, if the schedule holds up, become an RFC around the end of the year. To obtain further information on this work, contact J. Paul Holbrook (ph@sei.cmu.edu).

4.2.0.0.1 Disaster Recovery Plan Another security related policy which a site should prepare is the Disaster Recovery Plan. Many administrators think that only big commercial sites need a Disaster Recovery Plan because they have high demands for immediate continuation of the data processing after a disaster. But, in my opinion, even very small Unix sites should have a Disaster Recovery Plan. It need not to be be a solution with alternate sites or standby equipment, but every Unix System Manager needs to consider what to do if he or she has lost essential parts of the equipment, due to natural disaster, fire, theft, or just plain failure.

The purpose of the disaster plan is to have everybody at the site play "What if?". What if we lose the root disk on our main file server? What happens if our only tape drive fails and we cannot do any restores for another week?

The most important part of writing the Disaster Plan is to analyze the possible manner in which the site can fail, and what, realistically, can be done to prevent or recover from that situation. All too often a site has none of this in place and, when the disaster strikes, much time is wasted when managers and technical personnel alike scurry around as beheaded chickens.

In my experience, most UNIX sites' needs for disaster recovery can be provided by good off-site storage of backup tapes combined with a hardware maintenance service and/or stocking of strategic spare parts (like a replacement root disk drive).

5. Conclusion.

Good system administration requires careful planning of the approaches to be used. The use of well documented policies and procedures (like some of the examples in this paper) can help prevent the development of acute situations. As a result, the site will become more secure and the system more reliable. This will benefit both users and management but, most of all, it will benefit the system administration personnel, as the working environment will be less chaotic and stressed. This will help to increase the quality of the system administration provided for the site. Bear in mind, however, that no tool or policy can replace common sense.

REFERENCES

1. Dan Farmer & Eugene H. Spafford
   The COPS Security Checker System
   USENIX, Sommer 1990.

2. Bruce Spencer.
   Spy: A UNIX File System Monitor.
   USENIX Workshop Proceedings 1989 Large Installation System Administration

3. Bjorn Satdeva.
   On the Human Aspect of UNIX System Management
   Administration. In ROOT, volume 1, 1989, issue 4 & 5.

4. Bjorn Satdeva.
   A Lazy Man's Guide to UNIX System
   Administration. In ROOT, volume 2, 1990, issue 2

5. The Cuckoo's Egg
   Clifford Stool
   Doubleday, N.Y., 1989.

6. Patric H. Wood and Stephen C. Kochan
   UNIX System Security
   Hayden Books UNIX System Library, 1988

7. Patric H. Wood and Stephen C. Kochan
   UNIX System Security
   Hayden Books UNIX System Library

8. rdist(1).
   In 4.3 BSD - UNIX User's Reference Manual.
   Department of Electrical Engineering and Computer Science,
   University of California.

The purpose of the disaster plan is [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

REFERENCES

1. [illegible]

2. [illegible]

3. [illegible]

4. [illegible]
Clifford Stoll
Berkeley, [illegible]

5. Patrick H. Wood and Stephen G. Kochan
UNIX System Security
Hayden Books UNIX System Library, [illegible]

6. [illegible]

7. [illegible]

Networked UNIX* without the Superuser

Mark E. Carson, Janet A. Cugini, Sohail H. Malik§,
Mythili Kannan, Wen-Der Jiang

IBM 182/3F43
800 N. Frederick Ave.
Gaithersburg, MD 20879
...!uunet!pyrdc!ibmsid!{carson,cugini,malik,myth,jiangwd}

Background

Beyond the rights they inherit from the user who invokes them, trusted programs may have additional *authorizations*: different effective user or group ID's in the case of ordinary UNIX setuid, and one or more discrete kernel privileges, in the case of a Secure XENIX#[1]-like privilege scheme. We have recently been faced with the issue of how these added authorizations should be handled in a network environment in our work on a secure multilevel version of the X Window System.** Here, we briefly discuss the considerations we have taken and outline our approach.

A consequence of both setuid and privileges is that authorization is no longer a user characteristic (alone), but is rather a property of the (trusted) program being run, or of the combination of the user and program together. This is generally preferable, since splitting privilege from the user means, for example, that an administrative user cannot inadvertently give administrative powers to a Trojan horse program merely by running it. However, in a network, client-server environment, we have the question, how can a server be reliably informed of its client's authorizations? When authorizations are strictly a user characteristic, existing means of network authentication (such as Kerberos[2]) will also automatically transfer authorization information as well. However, since authentication schemes generally rely for their security on something provided by the user (such as a password), it's not immediately evident how they can allow transmitting augmented authorization information with any assurance. In other words, if privileges are not a user characteristic, the user's proofs of identity won't suffice to establish them.

Both schemes (at least in many implementations) also allow temporary lapsing or permanent dropping of the augmented authorizations when they are no longer needed or desired. But then, since authorizations are at least somewhat changeable (unlike a user's identity, which remains fixed for a login session), some provision has to be made for changing them in the course of a network session.

Our approach

The approach we have taken, although of general applicability, has been influenced by the nature of the X protocol. First of all, unlike for example NFS§§[3], the X protocol is used directly by ordinary user programs, which need not be trustworthy. Secondly, again unlike NFS, the X protocol is quite "stateful"; in fact, the X server's basic job can be viewed as maintaining state on behalf of its clients.

Hence we have moved initial establishment of client authorizations outside the X protocol. On the client side, when a client connects to a server port, the client machine's kernel (conceptually at least) squirrels away a copy of the client's current authorizations (all of which are conveniently part of the process structure). The server side similarly stores the

* UNIX is a trademark of AT&T.
§ VDG, Inc.
# XENIX is a trademark of Microsoft.
** The X Window System is a trademark of MIT.
§§ NFS is a trademark of Sun Microsystems.

Disclaimer: The work described herein is part of a research project. No IBM product commitment is made or implied.

server's current authorizations upon the server's accept to allow for "mutual authorization," although we don't actually make use of this for X. A new ioctl, SIOCGPEERAUTH, then allows inquiring about the authorizations of one's peer.

This much, the collection of the information on one side and the interface for asking after it on the other, is quite straightforward to implement. In fact, for UNIX domain socket connections it's basically the whole story. However, for TCP/IP domain sockets, the difficulties arise in moving the information from one side to the other in such a way that the other side can believe it. We entrust this job to "authorization servers"* on the two machines. There are two basic approaches to making this work.

In the simple case where the network itself is physically secure and all hosts are equally trusted and uniformly administered (the principal case we are concerned with now), the UNIX "privileged port" concept provides all the assurance we require. Each machine has an authorization server which binds to TCP port 113, the "authentication server" port mentioned in RFC 931.[4] The authorization server consists of a kernel part (structured as a device), which receives and fills requests, and a user-level part, which handles communication between its kernel part and its peer. The authorization servers communicate using a simple compatible extension of the protocol described in RFC 931.

For the more general case (with the network potentially insecure and hosts at different levels of trust), we maintain the idea of the per-machine authorization server. The authorization servers now rely on Kerberos authentication to establish their identities, which implies they must have Kerberos passwords which are hidden as well as anything on that machine can be hidden. Even so, each machine has responsibility for deciding which privilege claims it will accept from each other machine. There are endless variations on what considerations to make in doing so. We are currently considering a relatively simple design where the Kerberos server maintains an authorization database describing how to handle (e.g. accept, delete, translate) claimed identifiers and privileges.

After the initial connection, any changes in client authorizations are done through new X protocol requests (and hence such changes can only be "downward" adjustments). Since the X server is stateful, only changes in authorization information need to be sent — such information need not accompany every X protocol request. Making such requests "in-band" has the major advantage that the client can exactly synchronize them with its other X requests. It need not worry that a particular request was not made with the proper privileges in force because of Xlib or socket-level buffering. (This is unlike the Mitre "Trusted X" approach[5], where authorization information is handled by the underlying communications stream, and which required some restructuring of Xlib and the server os code to function properly.)

It is important to note that with our design, absent a specific client request, a client is treated by the server as having the authorizations established at the time of initial connection. If a client, for example, drops all kernel privileges after connection and does not similarly request they be dropped by the server, it will still be treated as privileged by the server. This may seem wrong at first, but in the case of X at least is actually the correct approach, "more least privilege" as it were: most of the privileged X applications we are dealing with (such as the window manager) do not need operating system privilege after they are finished with initialization, but do need window system privilege throughout. Hence, after initialization, they can drop the more "dangerous" (as dealing with permanent objects) operating system privilege, but retain their window system privilege.

References

[1] M.S. Hecht, M.E. Carson, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, V.D. Gligor, W.D. Jiang, A. Johri, G.L. Luckenbaugh, N. Vasudevan, "UNIX Without the Superuser," *Summer 1987 USENIX Technical Conference Proceedings.*

[2] Jennifer Steiner, Clifford Neuman, Jeffrey Shiller, "Kerberos: An Authentication Service for Open Network Systems," *Winter 1988 USENIX Technical Conference Proceedings.*

* Unfortunately, it appears this term has been used for a similar, but not identical concept; it's hard, though, to think of anything else which is as appropriate.

[3] Russel Sandberg *et al.*, "Design and Implementation of the Sun Network Filesystem," *Summer 1985 USENIX Technical Conference Proceedings.*

[4] M. St. Johns, *Authentication Server*, RFC 931, 1985.

[5] Jeffrey Picciotto, *Trusted X Window System, Volume 1: Design Overview*, Mitre paper MTP 288 vol. 1, 1990.

Hardening Anonymous FTP

Jeffrey D. Roth
DLA Systems Automation Center
U.S. Defense Logistics Agency
Columbus, OH 43216
jroth@dsac.dla.mil

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Logistics Agency, the Department of Defense, or the U. S. Government.

1. Introduction

BSD and derivative Unix File Transfer Protocol (FTP) server implementations support an anonymous FTP feature. If enabled, via the creation of an *ftp* account, it allows users on other systems to access the file system subtree headed by the ''home directory'' specified for that account. Server execution of a *chroot()* system call causes the anonymous user to be unable to name (and thus unable to access) the rest of the system's file hierarchy. The only other limitations imposed on the anonymous FTP user are those connected with the ownership and permissions associated with the directories and files in the ''public'' part of the hierarchy.

Traditionally, many organizations on the Internet have used this feature to make available to others computer-related items including software, research reports and the like, with some maintaining large permanent archives of public domain or freely distributable material. As network connectivity increases and the technology begins to penetrate nontechnical components of organizations, anonymous FTP could be used to support many additional functions requiring wide dissemination of data, including personnel (job announcements), purchasing (requests for bids and proposals), public information, publicity and product safety, among others.

The convenience of an anonymous FTP service is that it avoids the significant costs associated with service providers' creation and maintenance of user accounts, while providing a restricted environment in which the users can operate. Also, unlike mail or network news services, it is recipient-initiated and not a broadcast or multicast mechanism. But its attractiveness is counterweighed by questions regarding its ''safety.'' The 4.3BSD manual's description for the *ftpd* server, while including instructions on its setup, also states that ''the anonymous account is inherently dangerous and should be avoided when possible.'' This paper is an attempt to identify risks associated with anonymous FTP, and to suggest means by which those risks can be reduced or eliminated.

2. Risks

The range of potential problems associated with anonymous FTP use varies depending on whether the service is isolated to a system devoted to it alone, or provided on a more general purpose system. It is worth considering the risks of the isolated server first, then extending the discussion to the more general case, before attempting to determine what policies or server features could potentially reduce those risks.

Traditional justification for computer security measures has been to protect the secrecy, integrity and availability of data. A ''public'' file area should by definition contain only data in some sense authorized for general release. The risk of anonymous FTP with respect to secrecy is that data that should not be public could be placed in the public area, intentionally or unintentionally. As an extreme example, an organization unfamiliar with the anonymous capability might create an account for a user ''Fred T. Paulson'' and assign him a userid of ''ftp,'' making all of his files accessible to the world! More likely, a user could transfer privileged information to a public directory, without realizing the ramifications of its release.

Independent of data secrecy issues, providers and users of public files share a concern that what the users retrieve is what the providers submitted, and not something substituted by a potentially malicious third party. The risk of anonymous FTP with respect to integrity is that data placed in the public area could be changed, for example to replace a ''clean'' program with a version that contained a trojan horse.

Finally, for any service to be useful to its clients it must be available to them. The risk of anonymous FTP with respect to availability is that, intentionally or not, some users of the service could generate requests that by their rate or their nature cause service to be denied to others.

Additional risks are associated with any decision to provide an anonymous FTP service on a computer system that simultaneously supports other services, for example a general-purpose time sharing system. In such cases the risk of inadvertent release of privileged information is increased. Misadministration of the system could result in exposure of user accounts. For example, a thoughtless administrator's copying a password file (containing users' encrypted passwords) to the public area would expose users to potential attack by a ''password cracker.'' Or an unsophisticated administrator or user might not realize that the ''public'' files are accessible to anyone on the network, not just users of local systems, and copy some inappropriate files into the public area. Through administrative errors, the integrity of files outside the public area could be compromised. More likely still, the presence of the anonymous FTP service could expose the other users of the system to denial of service. For example extensive anonymous FTP traffic could seriously reduce the CPU time, or disk or network bandwidth available for other purposes, or (if the public area included files or directories that could be publicly written) anonymous file transfers could cause file systems to fill up.

3. Policies

The two major policy decisions that must be made (other than whether to support anonymous FTP at all) are whether to provide dedicated resources for the purpose, and whether to allow files to be written by anonymous users.

Dedication of a machine to anonymous FTP eliminates the risks associated with having other accounts on the system. Dedication of a particular network link (circuit) to the service eliminates the potential denial of network bandwidth to other uses. Failing these, an organization could dedicate some less expensive resources, such as one or more disk partitions, to the anonymous service (thus limiting the effects of full file systems). If an organization elects to support anonymous FTP on a non-dedicated system, additional decisions are required to protect other users. For example, the organization would likely want to restrict the use of the anonymous service under some circumstances, such as by time of day or relative to the load average.

Limits on write access to the public area can reduce the threat of inappropriate public release of information and better protect the integrity of files. For example it is easier to instruct one public archive maintainer regarding matters such as trade secrets and privacy than to similarly advise a cast of thousands. Most important, restrictions on write access make it possible for an organization to associate file creation and modification with a particular user and hold her or him accountable. If anonymous users are permitted to create or modify files in the public area, any such files are by definition modifiable by other anonymous users. For example, if pairs of users arrange to share executable programs with each other using an anonymous FTP account somewhere as a staging area, a malicious third party could easily replace their programs with virus-infected versions. Users could, of course, be educated to use additional checks to ensure the integrity of files (e.g. checksums, encryption) but it is not clear that such efforts would be effective.

Consideration of the policy alternatives helps clarify the dual-edged nature of anonymity. The feature that makes anonymous FTP most attractive is the lack of any requirement to authorize users in advance (no access control). The feature that makes anonymous FTP most dangerous is the inability to identify the user (no authentication). Positive identification of the (in such a case no longer anonymous) user, for example via a public-key certificate, combined with effective legal remedies against abuse, would likely provide sufficient protection. Such a mechanism does not yet exist on the Internet. The implication for an organization considering permitting write access to its public file area should be obvious.

4. Server

Whatever policies are chosen they must be enforced. The BSD FTP server does not incorporate mechanisms for enforcing policies particular to anonymous access. This makes the system vulnerable to administrator errors that could be avoided by incorporating checks and controls in the server. For example:

- The enabling of anonymous FTP could be done via an explicit ftpd option rather than implicitly via creation of an account with a ''magic name'';
- The ''recommended rules'' regarding directory and file ownerships and permissions contained in the BSD manual page for *ftpd* (for example the warning not to include a password file containing encrypted passwords) could be enforced by the server;
- A ''anonymous read-only'' option could be provided for the server, over and above directory and file permissions;

Additional options could be provided to reduce risks of denial of service and create an audit trail:

- Administrator specification of time of day and load-dependent limits on anonymous use could be supported;
- Extended logging could be supported, including logging of all anonymous accesses and recording all files read or written, with the extent of logging controlled via a log level option.

5. Conclusions

While an anonymous FTP account can be valuable for disseminating a variety of information, its appeal stems from the lack of access controls rather than from its concomitant lack of authentication. Its presence on a system can pose risks that threaten the secrecy, integrity and availability of information stored on it. Careful administrative policies and practices, especially isolation of the service and restrictions on write access, can reduce the risks. Further risk reduction could be obtained via modifications to the FTP server to cause it to check for common administrative errors, permit restrictions on anonymous use, and enable additional logging. Without effective authentication, however, the lack of user accountability is justification for denying any anonymous write access.

Internet Gateway Security Checklist

Jerry M. Carlin
Pacific Bell
2600 Camino Ramon
Room 3CS90
San Ramon, CA 94583
jmc@srv.pacbell.com

ABSTRACT

Securing access to the Internet is critical since system access via the Internet is as insecure as any direct-dial modem port on a computer. This checklist contains some measures taken at Pacific Bell to secure the machine being used as a gateway to the Internet. It was specifically developed for a SUN SPARC running SUNOS 4.1 but should be basically applicable to other systems.

Gateway security measures:

- Restrict FTP access using /etc/ftpusers. Minimum entries are those supplied with the system: root, daemon, uucp, sync, sys, uucp, bin, adm, audit, news, nobody and sysdiag.
- Create /etc/shells and put only sh, ksh and csh as entries to restrict users to those with valid shells.
- Either do not implement anonymous ftp or implement securely by using chroot including having only a minimal /etc/passwd and available commands.
- Do not implement tftp. The gateway has no need for it.
- Preferably do not run NFS. Any exported filesystem can be written to and setuid programs introduced. If it is critical to run NFS, write a program which looks for such files and run at least once daily. Also, disable NFSSERVER and allow NFSCLIENT only for a source only (incoming ftp) filesystem and put 'trusted' machines in /etc/fstab.
- No other machine should have the gateway machine in /etc/hosts.equiv. The gateway machine should not have any entries in /etc/hosts.equiv except perhaps for one that is used for rdist.
- Set /.rhosts so that no other machine is equivalent. There should be no ~.rhosts file in the gateway and no other .rhosts file should contain the name of the gateway.
- Verify that /etc/hosts has no ambiguous entries. This means that the fully qualified name is first and any aliases second rather than the reverse. Use DNS (domain name server) rather than having a large hosts file.
- Limit /etc/inetd.conf services. Disable tftp, finger, systat, netstat, mountd, rexd, ypupdated, sprayd, walld and rusersd.
- Enable only the minimum services (boot daemons) in /etc/rc.local. Since this machine will not be a server for diskless workstations, do not run any daemons that are only needed for such configurations.
- Change /etc/services to track changes in rc daemons. The minimum services are echo, discard, telnet and domain.
- Use the SUN "Secure RPC" protocol where ever possible.
- Run in.ftpd with -d (debug) option and set /etc/syslog.conf to log 'demon.debug' so that we can see possible ftp attacks. Make sure the demon logging file is not publically readable as passwords get stored therein.

- Do not assume that '*' in the password (passwd.adjunct) file forbids all access to that ID. Where possible also put /none for directory and /noshell for shell. If someone has set up .rhosts the '*' does not really keep someone off.

- Either do not run YP on the gateway machine or make it a separate domain. If you use YP, do not put any entries in yppasswd and remove the "+" in /etc/passwd. YP has security weaknesses since no rigorous authentication is done before accepting an update. These weaknesses may have been fixed in SUN 4.0.3 if you run 'ypbind -s'.

- Part of UNIX OS TCP/IP assumes "privileged" ports (< 1024). Non-UNIX machines, including PC's and protocol analyzers, do not pay attention to port numbers so the gateway should assume the remote machines ignore the concept of "privileged" port.

- Use IP filtering. Run gated(8) which supports restrictions on sending and receiving routes.

- Configure the Internet gateway router for maximum security. Do not allow the router to be used as a 'diverter' to hide calls. Ideally use the router's port-level IP filtering to deny service to disabled daemons.

- Filter out source-routed datagrams. Do not let the source machine tell your gateway how to forward a packet.

- Require that all machines that want connectivity to the central server install security patches as they are issued and maintain adequate security measures using the local UNIX security standards documents as a starting point.

- Check what terminal lines are "secure".

- Check what user id's are in groups wheel, kmem and bin.

- Mount all filesystems but /usr as 'nosuid'. Mount /usr as -ro and only mount -rw to install new software.

- Make sure *.netrc* files do not contain passwords to critical machines. Anonymous ftp "passwords" are OK.

- Remove compilers and loaders if not needed.

- Chmod all bin, lib and other critical directories to mode 711 or 710. That is make sure write and read are forbidden where possible.

- Watch out for a bug in some versions that use the domain name server to do a reverse lookup, mapping the internet number to a host name using the in-addr.arpa domain. (reported by CERT)

- Install a 20-minute timeout so that workstations and terminals are not left available if someone leaves them logged in.

- Install a no-trespassing notice to protect yourself legally.

- Develop, implement and document a contingency plan to pull the gateway to internal network ethernet connection if a significant security problem is uncovered.

- Do logging on another machine; set *loghost* to be another machine in /etc/hosts and configure /etc/syslog.conf appropriately.

- Monitor mail traffic to pick up attempts to find valid user names by using the SMTP port: *telnet foo@bar.com 25* followed by a series of *VRFYs.*

- Run a program which looks for *trojan horses* weekly.

- Run a program which verifies permissions and checksums at least weekly to audit which files and directories have changed, if any.

- Run a password cracker on a remote machine at least monthly.

- Run *COPS* periodically to look for problems.
- Run C2 auditing with flags "-ad,-lo,-p0,-p1" to look for failures in critical system calls that might indicate possible intrusion. Do an *audit -s* and *praudit* nightly to look for possible problems.
- Use Kerberos on the Internet gateway machine for connection to the central server.
- Run C2 security on the central server as backup security.
- Have no direct connection to production/mission critical systems. Instead, use a router that allows for restrictions on addresses and ports to connect to these systems.
- Run NNSTAT on another machine to track traffic on the gateway to see what is accessing the gateway.
- Use an intrusion detection expert system to minimize time required to examine logs.

References

[1] David A. Curry, "Improving the Security of Your UNIX System", SRI International, ITSTD-721-FR-90-21, April 1990.

Communicating Vulnerabilities: Perils and Pitfalls

David S. Brown
E. Eugene Schultz, Jr.
University of California
Lawrence Livermore National Laboratory

Abstract

System administrators have been faced with an increasing number of computer intrusions in recent years. A disproportionate number of these intrusions has involved UNIX systems. The Department of Energy's Computer Intrusion Advisory Capability (CIAC) team has become painfully aware that many system managers have left opportunities for instrusion because they have not closed vulnerabilities in their systems. These systems managers, however, often attempt to learn about UNIX vulnerabilities, but are unable to obtain the information they need. There is something wrong with the way vulnerability information is currently distributed: a more effective way of communicating vulnerabilities is badly needed.

Communicating vulnerabilities, however is not a simple issue. We have seen two extremes of approaches. One approach is what we call the broadcast approach, in which concerned parties inform the entire UNIX community about every known vulnerability. The second approach can be called the mute approach, in which vulnerability information is withheld from all concerned parties, even those who could help create a solution (e.g., vendors). Both of these extreme approaches have created problems for the UNIX security community.

We advocate a more moderate approach, specifically that we release vulnerability information to our constituency, peer response teams, and vendors in what might be called a "need-to-know" manner. We first attempt to determine exactly who wants the vulnerability information. Part of this initial process includes considering the reputation and reliability of the person requesting vulnerability information. We then consider whether the requester really needs the information, and, subsequently, how much information is needed to fix the problem. We often find that open discussion is needed to determine how we can share this information with trust and confidentiality to the minimum number of players who can secure current systems and create permanent fixes. We also use workshops on incident handling to provide specific information about vulnerabilities to groups of specially selected individuals.

We also advocate automated tools to assist system managers in detecting UNIX vulnerabilities. Tools such as Dan Farmer's, COPS (Computerized Oracle and Password System) and LLNL's SPI (Security Profile Inspector) are useful in this capacity. CIAC team members are also developing a patch checking tool for SunOS systems.

Security Breaches: Five Recent Incidents at Columbia University

Fuat Baran <fuat@columbia.edu>
Howard Kaye <howie@columbia.edu>
Margarita Suarez <marg@columbia.edu>

Columbia University
Center for Computing Activities
New York, NY 10025

Abstract

During a two-month period (February through March, 1990) Columbia University was involved in five break-in incidents. This paper provides a detailed account of each incident as well as what steps we took, both short-term and long-term, to reduce the likelihood of future incidents.

1. Site description

Columbia's computing environment is networked and distributed. Many departments have their own computing facilities — some have only a few workstations or small minicomputers, and others (like the Computer Science department) have dozens of workstations and several mainframes/super-minis. Departmental facilities are maintained in varying degrees by the departments that own them; the Computer Center provides the central network and tries to notify departments of possible problems. This is complicated by the fact that many departmental facilities are virtually unadministered.

The campus is interconnected via an Ethernet backbone, and through a central ROLM CBX. The phone switch allows people to connect to the campus terminal servers via phone lines, providing virtually unlimited on-campus connections. There are also 64 inbound modems, which can call any host connected to the switch. Unfortunately, the CBX ''hides'' the origin of inbound modem connections, making them difficult to trace.

Columbia has a variety of external network connections: Internet (via NYSERNet), BITNET, and CCNET (a DECNET network connecting several universities). While all of this connectivity is convenient, it also makes the campus an easy target for break-ins from outside the university.

The computer center (CUCCA) runs the network as well as the central academic and administrative computing environments. We have a cluster of three central Sun-4/280 server machines which are primarily used by the student body. We also run two Encore Multimaxes as staff machines. Most of the users of these staff machines are novice computer users, who use the systems primarily for electronic mail (EMAIL). These five machines make up our central UNIX systems. They are reached via two sets of terminal servers: the Suns are logged into via two cisco terminal servers, and the Encores via six Annex terminal servers. All of these systems mount each other's file systems via NFS. Sun users' home machines are randomly assigned within the cluster. The systems have a total user population of about 3000 users: 1500 students, 250 instructors, 1000 staff, and 200 guests. These academic systems are not used for anything sensitive, that is, there is no proprietary or confidential administrative information on them.

CUCCA also runs several IBM VM systems and a few PS2/Macintosh microcomputer labs. For internal staff use, we have several workstations of various types (Vaxstation II's, NeXT's, IBM RT's, Sun-4/110's) in our offices. The UNIX systems group has five members who support these CUCCA systems. There is also a User Services group of about twenty full-time consultants, as well as around 40 part-time student consultants who support the user community.

2. Incident Reports

During a two-month period (February through March, 1990) Columbia University was involved in five break-in incidents on various Computer Center systems. According to to the Computer Emergency Response Team (CERT) and *The New York Times*,[1] many other Internet sites were apparently experiencing similar problems during this time period.

2.1. Case #1 16-Feb-90

On Friday, February 16, 1990, at around 5 P.M. a member of our UNIX systems group noticed that one of our Multimaxes felt uncharacteristically sluggish for a Friday evening. A quick look at all running processes to try and identify what was using so much of the system revealed a program called *program* running as user *user1*.[2] Since *user1* was a guest account on our system and guests usually use their accounts to read mail, netnews, etc., and not for running CPU intensive programs, we decided to investigate.

A look at *user1*'s *ksh* history file to see what *program* was and where it was run from revealed unusual activity.[3] *user1* had connected to a directory named ".. " (dot dot space space) and had run *program* from there. This directory contained a copy of our */etc/passwd* file called *funlist*, and a list of 324 words called *list* — containing a lot of first names, names of famous people and teams, and

[1] "Computer System Intruder Plucks Passwords and Avoids Detection," March 19, 1990; "Caller Says He Broke Computers' Barriers to Taunt the Experts," March 21, 1990.

[2] Specifics such as hostnames, usernames, passwords, etc. will not be mentioned here, though *program* was the actual program name.

[3] The computer center's policy document states: "When necessary, the computing center full-time staff may access users' files. This will be done only for the purposes of maintenance and security of the system."

miscellaneous other words.[4] In this directory we found another copy of the executable file *program*, though the source code was not there. After examining the executable using tools such as *strings* and *nm*, we concluded that *program* was a "password cracker" (or perhaps a "password checker" — depending on your point of view).

```
$Header: pwchkr.c,v 1.2 85/11/30 22:42:07 richl Exp $
shell: %s
%s -- Problem: null passwd:
%s -- Trying "%s" on %s
Problem: GUESSED:
shell: %s passwd: %s
/etc/passwd
```

Figure 2-1: Some of the strings in *program*

```
_chkpw                          _checkgecos
_uandltry                       _checkcase
_try                            _chknulls
_setpwent                       _users
_endpwent                       _chkwords
_pwskip                         _PASSWD
_getpwent                       _EMPTY
_reverse                        _line
_verbose                        _passwd
_singles                        _Curpw
_backwards                      _Wordlist
```

Figure 2-2: Some of the symbols in *program*

Further inspection of *user1*'s *ksh* history file revealed that they had been reading our *crontab* file and the scripts invoked by *cron*. Also, we saw attempts by *user1* to connect to other users' temporary directories in */tmp*, and an attempt to propagate the *program* executable to other machines via NFS.

At this point we decided to turn off *user1*'s account by setting the login shell to */etc/turnedoff*.[5] We contacted our User Services group to inform them of our actions and found out that *user1*'s account belonged to a Barnard College staff member who was currently on maternity leave. Looking back through our accounting records we determined that *user1*'s account had been in use regularly starting on February 13, and that during this time it had accumulated a CPU charge for over 400 CPU minutes of usage, all running *program*. Login records showed that all logins to our machine were via one of our cisco Systems terminal servers.

Strangely, *user1*'s password was not in the list of passwords that we found. We later discovered that a few months prior to this she had forgotten her password, and contrary to our policy, a staff member

[4] a complete list of these words, plus other words found elsewhere, has been compiled by CERT

[5] */etc/turnedoff* informs a user at login that their account has been turned off for usage restriction violations, and gives them a phone number to contact to discuss reinstatement.

had reset her password to be her username, with the understanding that she would immediately change it to something else.[6] She neglected to change it, and according to our logs her password had remained the same until the break-in, when on February 14 and 15 someone, presumably the "cracker," changed it a couple of times.

After turning off *user1*, we examined the current day's logs, and we noticed a brief *su* process being charged to *user1* with a timestamp *after* the ID was turned off. From the tty name associated with the charge record we saw that this attempt was made from *user2*'s account, which had just been logged into, also from one of the cisco terminal servers. This was strange for two reasons: 1) *user2* was a guest account belonging to a former staff member whom we knew, and 2) the login was from a cisco terminal server rather than an Annex terminal server — legitimate users of our systems predominantly use the Annex terminal servers to get to this staff machine, while the cisco's are used by students to get to the student machines. A quick call to *user2*'s Wall Street office confirmed that he wasn't using his ID at the time. It turns out that *user2*'s password was a short, personal name which was in *list*.

We decided to watch this session for a while by tailing the shell history file and saw *user2* trying to access the files in *user1*'s ".. " directory, as well as wandering around trying to get into other guests' home directories. He then tried to send mail to an off-campus site. We intercepted this message when an incorrect address caused it to bounce, and we removed the bounce notice. In this message, which was signed with a pseudonym, *user2* informed a friend that this ID was being used without the owner's permission.

We traced the connection from the cisco terminal server to a 2400 baud inbound modem on our university CBX attached to a trunk line to New York Telephone. Tracing it further would have required the cooperation of our Telecommunications department as well as New York telephone. We have since found out that getting NYTel to do a phone trace would be extremely difficult (if at all possible) and time consuming due to New York State laws restricting caller identification.

To determine which other accounts besides *user1* and *user2* may have fallen into the hands of the "cracker" we decided to run the word list against our rather large password file. We split the task up and ran a total of 12 password crackers simultaneously on several Sun-4/280's and Encore Multimaxes. We completed this by 3 A.M. and identified 16 accounts which had passwords in the list. We modified the shells on all these ID's to */etc/badpasswd*, which informed the users at login to go to our Business Office to get their ID's turned back on after verification of their identity. We then turned off *user2* as well.

According to our weekend logs, there were over 25 failed attempts to login as *user1* and *user2*, which leads us to suspect that information on break-ins was being widely disseminated — it did not seem

[6] Our */bin/passwd* won't let users pick "dumb" passwords, but another utility allows staff to set arbitrary passwords.

like one person would stubbornly try the ID so many times after getting a message that said the ID was turned off. At one point on Saturday someone claiming to be *user2* even called up a consultant in one of our terminal rooms asking for information on access to ''anonymous email.''

Details of this break-in were provided to CERT as they became known to us.

2.2. Case #2 18-Feb-90

During the weekend following case #1 the UNIX Systems group performed periodic spotchecks to see if there was any other unusual activity on the systems. On Sunday, February 18, 1990, we noticed a student, *user3*, on one of the instructional Suns running some CPU intensive processes. Also, these programs were being run with usernames as arguments — in particular, familiar staff names (which are at the top of the passwd file). A quick check of this user's shell history and directory revealed a password cracker which used a 215 word list.

By looking at this user's other processes, we suspected that it was the account owner himself that was running the password cracker (he was doing his homework while running the cracker in the background). We contacted him by phone, at which time he admitted to running the password cracker. His account was invalidated, and after a meeting with the Manager of User Services his case was referred to his dean. Though the student did not think he had done anything seriously wrong, his dean held a disciplinary hearing, and, on our recommendation, placed the student on probation. The student's account has since been revalidated.

At the meeting with the Manager of User Services, *user3* said that he had gotten the idea to break passwords partly through a microcomputer bboard message, which was written by someone whom he didn't know personally who was using a pseudonym. He had also corresponded about password cracking with a friend at a university in the United Kingdom.

2.3. Case #3 20-Feb-90

On Tuesday, February 20, 1990 (two days after case #2), another guest account, *user4*, belonging to a former university employee was using an unusual amount of CPU cycles running a program called *square*. The home host for *user4* was the same machine that *user1* and *user2* were on, though *user4* was logged in on one of the student Suns and accessing her files remotely over NFS. Due to the events of the past week, we had been on the lookout for such processes and immediately became suspicious. Looking at *user4*'s shell history file revealed unusual activity. Her home directory contained many encrypted files, and she was decrypting them one at a time and using them as input into the program *square*. The only plaintext file was the one currently in use. We examined it and it turned out to be a passwd file for some foreign system — the *gecos* field of some users had 812 area code phone numbers (Indiana). We later confirmed with Indiana University that the file was indeed a copy of the passwd file to a system there which had recently been broken into from a site in Texas.

```
_pw_name
_crypt
_fgetpwent
_pw_passwd
_input_file
_strcmp
Enter number:
%d squared equals %d
Enter your guess:
Correct is %s
```

Figure 2-3: Some of the strings in *square*

We also found a plaintext C source file called *getch.c* which contained the sources to *square.* This program prompted for a magic number (32768) and when it got anything else it just printed out the square of the number and exited. When given the magic number, it opened a file (with the name hardcoded, which is probably why the source file was in plaintext, since it had to be recompiled for each passwd file), did an *fgetpwent* on the descriptor, prompted for a password guess and encrypted it for each user in the passwd file trying to see if it was the password for any of the users.

We decided to turn off *user4* (by setting the login shell to */etc/turnedoff*) and went through our logs and through the various files in *user4*'s directory. We saw recent activity on this account, which had previously been pretty much unused. The *.newsrc* file here revealed that the various *comp.bugs.**, *comp.dcom.**, *comp.protocols.**, and *comp.unix.** newsgroups, as well as *soc.culture.china*, *soc.women*, *alt.hackers*, *alt.cyberpunk*, *comp.virus*, and *misc.security*, were read. A few days after the ID was turned off, we found queued mail (deferred because the destination was temporarily unreachable) to someone in response to a *comp.dcom.telecom* article referring to a phone hackers' newsletter called *Phrack*.

While we were looking at *user4*'s previous activities, we found a copy of one of the data files and the program *square* in */usr/tmp*. This file was owned by *user5*, a student account from the past semester. Looking around some more we found a file called *.newshell* in *user4*'s home directory, which was a world executable copy of */bin/sh* that was setuid *user4*. *user5* had apparently used this to access the files in *user4*'s directory. Immediately after a failed attempt to login to *user4*, we saw a login to *user5*, so we suspected that the same person had broken into both accounts. The shell history file shows *user5* connecting to *user4*'s directory and running *.newshell.* He also *grep*'ed for *user4* in */etc/passwd* and saw that the shell had been changed to */etc/turnedoff*. We subsequently turned off *user5* as well. To date, *user4* has not gotten in touch with us about her turned-off account, and *user5*'s has since expired.

As always we informed CERT about all that occurred. The encrypted files in *user4*'s directory had short names that sounded like possible hostnames, so with the help of *nslookup* we tried to determine the fully qualified names and made a list of potential sites. We sent mail to people in charge of those systems warning them that their passwd files might have been stolen to be processed by a password cracker.

One of the system administrators we contacted suggested we use *Crypt Breakers' Workbench* (CBW) to decrypt the files we found. We compiled CBW, but did not have the time to learn how to use it properly.

2.4. Case #4 22-Mar-90

On Thursday, March 22, 1990 (after a quiet month) we received electronic mail from a system administrator at the University of Virginia telling us that they were experiencing break-ins on their system originating from two of our machines. One was a Sun-4 student system, and the other was a cisco gateway. We contacted the system administrator by phone and obtained more information. It turns out they had been informed by another university of break-ins originating at Virginia which Virginia traced to a guest ID on their system. This ID had been logged into from our machines on several occasions, at various times, mostly early in the morning between 3 A.M. and 7 A.M. Virginia provided us with exact login times for the most recent occurrences, one of which turned out to be at 7 A.M., when exactly one student, *user6*, was logged in on the machine in question.

user6's directory did not contain anything interesting other than the fact that it lacked a shell history file. The user's shell profile did not rename the history file (via the *HISTFILE* environment variable), nor did it do anything fancy to automatically remove it at login or logout, which led us to suspect that it had been manually deleted. One other interesting item was that the logins were on a machine other than *user6*'s home machine and had occurred regularly over a two-week period of time, almost always from our cisco terminal servers. We noticed regular logins on the home machine that ended approximately one week before we were notified of the break-in. Login records also showed a few logins (on the non-home system) from Virginia at times which we later confirmed coincided with break-in times at Virginia.

It is interesting to note that the particular machine at Virginia that got broken into had a name that was also the name of one of the encrypted files we had found in *user4*'s directory (see case #4 above). We mentioned this to the administrator at Virginia who said he had heard that his passwd file had been stolen a while ago.

We turned off *user6*'s account, and that afternoon there was a single login attempt made on it. A week later the student who owned the account got in touch with us in response to the message he got when he attempted to login. We set up a meeting with him and the Manager of User Services. He told us that he had been on vacation in Texas during the past week (spring break) and that he didn't even know he could login to any machine other than his home system. He was apparently a novice computer user taking his first CS class. We asked him what his password was, and it turned out to be the name of a musical group. We believe it may have been guessed by a password cracker (though this particular name was not in any lists we had seen). We subsequently reported this incident along with the guessed password to CERT (and his ID was turned back on with a new password).

2.5. Case #5 26-Mar-90

Four days later, in the evening of Monday, March 26, 1990, a member of our User Services group noticed what appeared to be a runaway *mm* process.[7] This sometimes happens when a user gets logged out from a terminal server and the host end does not clean up properly. To determine if the user was still logged in he *finger*'ed the Annex terminal server that this person had logged in from. On the Annex he noticed a suspicious outbound *telnet* connection (to a foreign IP address), and by *finger*'ing the IP address found that it belonged to an Annex terminal server at an east coast university. The only session on that Annex (with our Annex as the source IP address) had an outbound *telnet* connection to a cisco gateway at another east coast organization. Checking on *that* gateway, again with *finger*, showed an outbound *telnet* connection to a host at the University of California, San Francisco (UCSF).

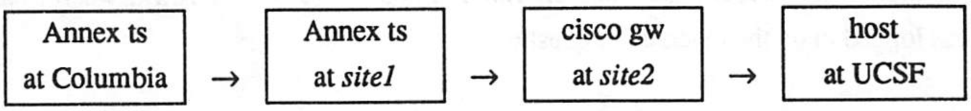


Figure 2-4: multi-hop connection to UCSF

Using the *tap* feature of our Annex terminal server, we were able to silently monitor all traffic on the line that this person was using, and logged it to an *xterm* log file. We watched as he logged into an account at UCSF, checked up on the logged in users (*w*), running processes (*ps -eafg*), mail for the user he logged in as (*cat*'ed */usr/spool/mail/user7*), another *w*, *cd*'ed to */etc* and got a directory listing. He then *cat*'ed the *hosts.equiv* file and *grep*'ed for "gw" (gateway) in the */etc/hosts* file. After another *w*, he *grep*'ed for various users in */etc/passwd*. After one final *w*, he disconnected.

We spent the next hour or so trying to get in touch with someone at UCSF in order to report the suspected break-in. We tried getting a phone number using the Domain Name System (DNS) *Start of Authority (SOA)* records as well as the *whois* database at *nic.ddn.mil.* Mostly we reached "phonemail" systems, since it was after business hours. We left a few messages and continued trying to reach someone in person. Eventually, we *finger*'ed all the sites in the *hosts.equiv* file we saw and found a phone number for an operations account. We called, and the night supervisor got in touch with the system manager for the broken-in machine, who called us back.

We wrote up a report of the incident and sent copies to CERT, UCSF, and the two intermediate sites that were involved. One site responded the next day, saying that they had just put a password on inbound connections to the gateway, while the other never responded. When we checked a month later, nothing had changed on their Annex.

[7] *Columbia-MM* is our locally written mail user agent.

We turned terminal server access restrictions back on (see section 4.8). They had been temporarily disabled due to hardware problems on the security host which the Annex would query for access restrictions.

3. Summary of the Five Cases

3.1. How They Got In

The illicit access to our systems occurred predominantly over dialup lines. Connections were made from our inbound modems to both Annex and cisco terminal servers, from which the crackers logged into our large UNIX systems (Encore Multimaxes, Sun-4/280's). In one case, both dialup lines and remote *telnet* sessions from another site were used (see case #4).

In all cases the accounts were broken into by password guessing. Despite our password selection policies, some people were able to get around the restrictions (either intentionally in the case of staff, or unintentionally — they forgot their password and a staff member changed it to something guessable, like a last name). In other cases, the dictionary of disallowed passwords we were using didn't have all the words in the cracker's list. No software security holes (e.g. *sendmail, ftpd*, etc.) were used. This may be because we try to fix known security holes as soon as we find out about them, or it may simply reflect on the type of break-ins that we saw. We suspect that once a user/password pair was cracked, this information was often being shared by a large number of people (see case #1).

Of the accounts that were broken into, two belonged to former staff members (now guests), one to a guest from an affiliated organization and two to students (one an ID from a previous semester and about to expire, the other that of a current student — though the illegal logins were on a non-home system). In all cases the accounts broken into were not being actively used.

The student in case #2 was using his own account in an attempt to crack passwords on our system (while simultaneously doing his homework).

3.2. What They Did

In three of the five cases we caught people running various forms of password crackers. None of the programs were especially sophisticated, and all the programs were written by the cracker or easily obtainable elsewhere. The program in case #1 seemed to be the best of the lot. It seemed to check the *gecos* field (to get first and last names, uppercased and lowercased), and used a list of passwords to try against our whole password file. The password cracker in case #2 tried a list of passwords on a single user (specified on the command line). The program in case #3 used a hardwired filename, accepted a guess from the cracker, and tried that on all password entries in the file.

In case #4 the account on our system was used to *telnet* out to remote sites. We found no evidence of any other activity (though history files were not available for the past two weeks' activity) and accounting data we had showed nothing other than *telnet*. In case #5 only our Annex terminal server was used. We were the initial network access point for a multi-hop break-in.

In case #2 we found mail files in which the concept of password cracking was being discussed with a friend. In cases #1 and #3 the cracker made use of electronic mail to either get in touch with a friend or try to obtain information heard on usenet.

Case #	Ran Password Cracker	Guessed Local Password	Used Us As Intermediary
case #1	our /etc/passwd	2 guests	no
case #2	our /etc/passwd	no	no
case #3	foreign /etc/passwd's	1 student, 1 guest	used our CPU
case #4	no	1 student	to/from Virginia
case #5	no	no	used Annex ts

Figure 3-5: Summary of cases

3.3. How They Were Detected

A major component in detection can be attributed to pure luck — being on the right system at the right time, noticing uncharacteristic sluggishness in system responses and investigating the cause, seeing suspicious processes being run by guests, etc. Accounting records, which aren't normally reviewed on a daily basis, also contained corroborating traces of break-ins once we consulted them. In the later cases, past problems had alerted us, so we were more suspicious and checked the systems more often.

Once we started investigating a particular user, we were able to find plenty of traces in the form of accounting records and files left around (''hidden'') in various places. None of the people in the above five cases had access to anything other than a couple of student or guest accounts (i.e. no *root* break-ins), so they were unable to cover their trails by modifying system logs and accounting records. The best that they did was to encrypt all incriminating files and delete source code, but even then, one or two files were left around in plaintext.

4. Lessons Learned

We have always been concerned about the security of our systems, especially since the university has such a large and transient user community. However, this rash of attacks on our systems has taught us some important lessons, and as a result, we have discovered ways to make our systems even more secure. We would like to share some of the methods we have implemented to try to prevent these attacks, as well the stop-gap measures we took when we noticed crackers on our systems.

4.1. Turn Off Abusers

CUCCA has a policy that disallows sharing of accounts. This means that if we detect that someone other than the owner of an account is using that account, we immediately turn it off. We have this restriction precisely so that we will be able to track down abusers and intruders. Our usual method of dealing with possible abusers is to set their shell to */etc/turnedoff*, change the mode of their home directory to 0, and kill their processes. The program */etc/turnedoff* displays a message telling the user who to contact to discuss getting the account turned on. That way we can verify the identity of the user in person.

4.2. Turn Off Potential Targets

In addition to turning off the obvious abusers, we found it necessary to turn off the potential targets — such as guest accounts with guessable passwords and all root IDs — and have the users come to us in person to have their passwords changed. We took advantage of this opportunity to reinforce security-consciousness in people who had privileged accounts.

4.3. Monitor System Activity

When we noticed the first breakins, there was a period when we found ourselves staying up late to monitor system activity. Some of this ''paranoia'' paid off: we were able to catch *user3* and *user4* because of their excessive CPU usage late at night, a time when few staff are usually logged in.[8] Unfortunately, there is a limit to how long a system administrator can stay awake, or type ''ps -alxww,'' for that matter. Luckily, our nights of vigil ended when we had apparently plugged enough holes, and the cracker activity subsided.

4.4. Eliminate Stupid Passwords at the Source

One precautionary measure we had taken was to modify the */bin/passwd* program to check new passwords against a dictionary[9] and strings in the *gecos* field, as well as for length and robustness. The BSD *passwd* program has a rather arbitrary set of criteria for determining ''richness'' of character composition, in addition to a well-known workaround that lets you set any password you desire, if you insist.

We created a new module called *esp*, for ''eliminate stupid passwords.'' This library is linked in with all password-changing user programs. By disallowing usernames, personal names in the *gecos* field, and dictionary words as passwords, we ''force'' people to choose better passwords. In addition we require passwords to be sufficiently long (at least six characters) or ''rich'' — containing digits, mixed-

[8] See case #2 and case #3.

[9] In fact, the comments in the BSD *passwd* program mention that this should be done.

case, and special characters.

This measure was in some cases defeated when a staff member used a utility which did not have *esp* linked in, but we believe that since the few cracked passwords were guessable, such password checking has protected the majority of our accounts from casual attacks. CERT was able to crack about 300 more accounts running their password crackers on our passwd file, and we have since incorporated their word list into our password checker. Currently we are using a dictionary of about 52,000 words. The list consists of standard English words from the *ispell* dictionary, words known to be used by password crackers (from the list given to us by CERT), and site-specific words such as hostnames. The code was borrowed from the *ispell* sources, and it uses a hash table, so it is relatively efficient. While some may argue that such restrictions leave no ''good'' passwords, changing one character to uppercase is usually sufficient to make an otherwise guessable password be acceptable.

4.5. Force Users to Change Stupid Passwords

Since *user1* and *user2* had both been broken into because of easily guessable passwords, we wanted to be sure that all guessable passwords were changed before the crackers got to them. In the beginning, we had been turning off accounts as we found them to have guessable passwords. This was necessary because we had no way of knowing whether the accounts had already been cracked or would soon be cracked. We set the shells of such accounts to */etc/badpasswd,* which was essentially the same as */etc/turnedoff*. The */etc/badpasswd* program simply displayed a message saying that the user's password needed to be changed, and that they should visit our Business Office in person to get it changed to a better password. This worked all right for the first few password changes, but it was getting a bit tedious for the Business Office to have to deal with so many users — these password changes had to be done by hand, and the users had to come in person and present their Columbia University Identification card. After we received a list of more than 300 accounts whose passwords were broken by CERT, we implemented a simple password changing shell to automate this process, thereby sparing the Business Office a deluge of password changers.

We have a convention of setting the password of new accounts to be the user's Columbia University Identification Number (CUID), so that they will know their initial password without us having to write it down. Obviously, in a university environment it is not unlikely that one can learn someone else's CUID number, so we require the user to change their password immediately when they first log in. The program */etc/ksh.first,* which is the shell given to newly created accounts, prints a message about the importance of a good password, asks new users to change their passwords before resuming normal login, then finally resets the shell to be */bin/ksh.*

In the event that the user has been found to have a bad password, we set their shell to our new program */etc/ksh.badpasswd,* which is similar to */etc/ksh.first.* This program prints a message notifying the user that their password is easily guessable and that they should change it to a better password. Since we cannot be certain that the person trying to log in is not a cracker, we have them pass a simple identification verification test before allowing them to change their password and log in: we ask for their CUID number. All attempts to log in using this program, whether failed or successful, are logged, and the log is reviewed periodically to check for repeated failed attempts (to check for CUID-crackers!). We have found this method to be quite adequate because it does not require that the user wait until business hours to get their password changed.

The shells */etc/ksh.first* and */etc/ksh.badpasswd*, as well as the */bin/passwd* program, were modified to use the new 52,000 word dictionary (which incorporated CERT's list). Obviously, no dictionary can be big enough to include all the words a cracker could try, but eliminating the obvious ones makes it a little bit harder for a cracker to break in.

4.6. Hide Passwords

We finally bit the bullet and implemented a shadow password scheme to hide the encrypted password strings from would-be attackers. In three cases, the cracker used our machines as a CPU to crack password files from ours and other hosts.[10] By making encrypted password strings available only to the superuser, we make it more difficult for someone to break into our machines by simply stealing a copy of the password file, and then crunching it at their leisure on some other machine. As an added feature, we placed randomly generated strings into the public password file to let potential crackers waste time trying to decrypt fake passwords. This allows us to watch for password crackers because a noticeable amount of CPU time would be used trying to break a fake password file.

Switching over to shadow passwords was fairly straightforward. We modified the *getpwent()* family of routines to check the effective *uid* of the process — the real password string can only be retrieved by root. Programs like */bin/passwd* update both the flat */etc/passwd* file and the shadow password file, */etc/shad/passwd*, placing the fake password string in the flat file, and hiding the real password string in */etc/shad*, a directory readable only by the superuser.

We obtained a copy of the sources to Berkeley's implementation of a shadow password scheme. However, due to the size of our password file, Berkeley's method was extremely slow, so we reimplemented it using a hashed database. The improved code has been sent back to Berkeley and will hopefully be available in their next release.

[10] See case #3.

4.7. Check for Bad *.rhosts* Files

To be sure that crackers could not use accounts gained on our machines to hop over to other sites and vice versa, we ran a special spot-check of users' *.rhosts* files. We wrote a simple *perl* script which checked for lines in users' *.rhosts* files which contained machines outside of the Columbia domain or which included a username different from this user, and recorded bad lines along with the owner in a log. After scrutinizing the output, we then moved "bad" *.rhosts* files to be *.rhosts.bad*, and sent mail to each user informing them of the change and giving an explanation of safe *.rhosts* files.

This is especially important for staff accounts, since staff are likely to have accounts on many different hosts (e.g., at other departments, or even at other universities). We would not like to see staff accounts compromised because of poor security elsewhere.

4.8. Turn On Gateway and Terminal Server Security

Our terminal servers and gateways had been used by the crackers in a couple of cases as "invisible" routes to other machines. We implemented passwords on each of our terminal servers and gateways to prevent this happening in the future.

A password is now required in order to telnet into our gateways and terminal servers. Also, we allow access only to local hosts, in order to prevent intruders from using these machines to hide their origins as they try to break into hosts outside our domain.

4.9. Report the Problem

As always, we kept close contact with CERT during this whole ordeal. We warned system administrators at other sites as soon as we realized they might be affected. In order to determine who the system administrators were at other sites, we used tools such as *whois*, *nslookup*, and *finger*, as well as the SMTP *VRFY* and *EXPN* commands.[11]

4.10. Educate Users

During this time of emergency, we tried to maintain close communication with our user community. In system bulletin boards, electronic mail, and the message of the day, we kept users informed of what was happening, explaining the importance of keeping their passwords secret and unguessable, warning them of unsafe *.rhosts* files, and apologizing for the inconvenience of having their userids turned off. We also tried to point out ways the users themselves could help detect if there had been breakins on our system: keeping an eye out for strange files in their directories, using *last* to verify the time they last logged in, etc. In addition to user education, we did a lot of staff education, as it is most

[11] See the appendix for a sample session where we look up some system administrators in the *columbia.edu* domain.

important that users with privileges (such as being in the wheel group or having a root id) have secure passwords and *.rhosts* files. Also, staff are likely to maintain several accounts on different machines, and therefore they are likely to have little-used accounts which they should take care to keep secure. All in all, staff userids are likely to be enticing targets for malicious intruders.

5. Loose Ends

5.1. Who Were They?

Most of the attacks which we suffered were rather anonymous. People didn't break in from other hosts, but through dial-in lines and terminal servers. Those that we did trace didn't use their own names, but stuck with pseudonyms. Data files which were left around were, for the most part, encrypted. When we were able to watch a break-in attempt, we did not seem to be confronted with expert UNIX "crackers." Often, MS-DOS style commands were used, and then followed up by their UNIX equivalents. Mail sent from penetrated IDs bounced: simple precautions were not taken to take care of little details like these. These initial attacks seemed to be followed by a flurry of similar attempts.

On the other hand, other attacks left dozens of encrypted password files from other sites sitting on our systems. These attacks seemed to be more careful, though history files were still found. We made a quick attempt at decrypting these files using the *Crypt Breakers' Workbench*, but as we had had no experience with the program, and since we already had a fair idea of the files' contents, it did not seem worth the time.

From the methods used and the traces left behind, as well as the repeated attempts to use the same entry points, we got the impression that we were dealing with several gangs of fairly unsophisticated crackers. In several cases crackers called attention to themselves through excessive CPU usage during hours when staff members were logged in. Alternatively, one cracker, trying to obscure his path, used our machine to reach other sites during a time when he was the only logged in user, making him especially easy to trace from login records.

In the end, all we could do was pass on what little data we had about these crackers to CERT, and hope that it would correlate with other information they might have. The only "cracker" who was identified was a local student, and he did not successfully crack any passwords.

5.2. Caller Identification

We ran into problems trying to trace the crackers via the telephone lines. First, we had to get past our CBX to find which New York Telephone lines were receiving the calls. Then, even when we had gotten that information using our CBX console software, we were unable to get telephone traces made. The university was in touch with NYTel about phone traces, but were eventually told that a court order was needed for any trace, and that the line the trace was being requested on had to be in an individual's name, rather than an organization's. Clearly, getting our dial-in modem numbers traced would be next to impossible, though our telecommunications department is still trying to work something out.

6. Futures

6.1. Preventative Measures

System administrators charged with the task of maintaining a secure working environment for their users need to keep up-to-date on potential security holes. This requires them to have access to channels that disseminate information on system security, such as security mailing lists, newsgroups, and other publications. Likewise, vendors need to be responsible to their customers by keeping current on these issues and providing timely fixes to any security holes that may be uncovered.

System administrators should take an active role in insuring the integrity of their systems. Periodic audits of the system as well as routine examination of log files can go a long way in detecting problems when they occur, and helping to minimize damage. Once a problem and its solution have been identified, fixes should be installed in a timely fashion. Perhaps to encourage this, once a fix is widely available it should be publicized. Methods of applying pressure on vendors should be developed to encourage them to provide fixes as soon as possible. How to do this is a sensitive issue that needs to be examined in depth. There is a fundamental dilemma in that it seems to require publicizing security holes to get vendors to take action, though this means that the holes can be abused in the interim.

We need better organization. Perhaps a single organization (like CERT) should coordinate contacts with law enforcement and telephone organizations. Perhaps there should be a distinction between tracing data and voice telephone calls. The laws for getting calls traced differ across state borders. Similarly, the procedures for getting traces are different in different places. The contacts and experience necessary for getting traces started can be more easily developed by a central organization than by individual institutions.

We need better cooperation throughout the Internet. When break-ins here pointed at other sites, it was often difficult (and merely through chance) that we were able to find administrators (and their phone numbers) at the other sites. The *whois* database should be expanded to contain accurate and up-to-date information. Several contacts and their emergency numbers should be included for each site. An automated periodic review process could get in touch with the listed contacts to verify entries.

In the event of network connectivity problems, the DDN Network Information Center may not be reachable. The information contained there should be replicated at several places; perhaps the database should be set up as a distributed database along the lines of DNS where each domain would maintain its own entries.

6.2. Ethics

Society needs to develop an awareness of the moral and ethical issues of computer usage. Breaking into a system should be judged with the same set of values as breaking into someone's home. Both are violations of privacy and cause severe distress as well as possible monetary damage to the victim. Just as burglary when committed by teenagers is not considered a ''childish prank,'' breaking into a computer system, regardless of how inadequately protected it may be, should not be considered the harmless activities of a precocious youngster. Movies such as *War Games* which glorify crackers ultimately do the community a disservice.

The Internet has grown from a relatively small community of trustworthy citizens to a cosmopolitan network with its share of vandals and other unsavoury individuals. We should try our best to educate the community as well as to deploy preventative measures to minimize the damage in the event of the inevitable attack.

7. Acknowledgements

We would like to thank Melissa Metz for helping during the incidents; our colleagues for proofreading this document and making extensive comments and suggestions; Benjamin Fried for *tap*'ing in Case #5; the members of CERT for moral support and processing our password file, and the members of our Telecommunications Department for contacting New York Telephone.

Appendix A. Locating a System Administrator

Below are excerpts from the transcript of a session in which we try to determine a night-time or emergency phone number for a system administrator within the *columbia.edu* domain.

First we query the NIC *whois* database for information on the *columbia.edu* domain.

```
% telnet nic.ddn.mil        # You can also use the whois cmd
Trying...
Connected to nic.ddn.mil.
Escape character is '^]'.
*  -- DDN Network Information Center --
*
*  >>> Our new address is 192.67.67.20.
*  >>> Service on our MILNET address (26.0.0.73) will end June 1.
*
*  For TAC news, type:                      TACNEWS <return>
*  For user and host information, type:     WHOIS <return>
*  For NIC information, type:               NIC <return>
*
*  For user assistance call (800) 235-3155 or (415) 859-3695
*  Report system problems to ACTION@NIC.DDN.MIL or call (415) 859-5921

 SRI-NIC, TOPS-20 Monitor 7(21242)-4
 The system will go down Thu 17-May-90  6:00pm until Thu 17-May-90  9:00pm
 for For Preventive Maintenance
@
@whois columbia-dom              # columbia domain
Columbia University (COLUMBIA-DOM)
   450 Computer Science
   New York, NY 10027

   Domain Name: COLUMBIA.EDU

   Administrative Contact, Technical Contact, Zone Contact:
      Cattani, Robert  (BC14)  cattani@CS.COLUMBIA.EDU
      (212) 854-2736

   Record last updated on 10-Aug-88.

   Domain servers in listed order:

   COLUMBIA.EDU                  128.59.16.1, 128.59.32.1
   HARVARD.HARVARD.EDU           128.103.1.1
   CUNIXE.CC.COLUMBIA.EDU        128.59.40.143

Would you like to see the known hosts under this secondary domain? n
Whois:
@logout
 The system will go down Thu 17-May-90  6:00pm until Thu 17-May-90  9:00pm
 for For Preventive Maintenance
Killed Job 19, TTY 152, at 14-May-90 17:34:40
 Used 0:00:04 in 0:01:37
Connection closed by foreign host.
%
```

Since it's after hours, calling the phone number given may reach an answering machine. Also, since the record hasn't been updated in two years, the number may be wrong. Using the information from the *whois* database we *finger* the contact address to see if his finger plan has an emergency phone number listed.

```
% finger cattani@cs.columbia.edu
[cs.columbia.edu]
Login name: cattani                     In real life: Bob Cattani
Directory: /u/cs/cattani                Shell: /bin/ksh
Last login Mon May 14 08:34 on ttyp7 from williamsburg.cs.
New mail received Mon May 14 19:22:00 1990;
  unread since Mon May 14 18:15:08 1990
Plan:
Robert Cattani                          EMail:  cattani@cs.columbia.edu
450 Computer Science
Columbia University                     Phone:  212-854-8107 (office)
New York, NY  10027

Computer Science Department
        Director of Research Facilities

%
```

We may not have found what we are looking for yet (this is another office number), so we try the Domain Name System (DNS) next, looking up a *Start of Authority* record.

```
% nslookup
Default Server:  cunixf.cc.columbia.edu
Address:  128.59.40.130

> set querytype=SOA
> columbia.edu.
Server:  cunixf.cc.columbia.edu
Address:  128.59.40.130

columbia.edu    origin = columbia.edu
     mail addr = hostmaster.columbia.edu
        serial=182, refresh=3600, retry=1800, expire=3600000, min=3600
> ^D
%
```

The mail address (*hostmaster@columbia.edu*) specified in the SOA record looks like an alias, so we will try to expand it using the *SMTP EXPN* (or *VRFY*) command.

```
% telnet columbia.edu 25      # port 25 is for SMTP
Trying...
Connected to columbia.edu.
Escape character is '^]'.
220 columbia.edu Sendmail 5.59++/0.3 ready at Mon, 14 May 90 20:35:42 EDT
expn hostmaster
250-<hostmaster@cunixf>
250-<alan@cunixc.cc.columbia.edu>
250-<chris@cs>
250 <cattani@cs>
quit
221 columbia.edu closing connection
Connection closed by foreign host.
%
```

The *hostmaster* alias on *columbia.edu* expanded to a few users and another *hostmaster* alias on *cunixf*. We try expanding the alias there next.

```
% telnet cunixf.columbia.edu 25
Trying...
Connected to cunixf.cc.columbia.edu.
Escape character is '^]'.
220 cunixf.cc.columbia.edu Sendmail 5.59/FCB ready \
                      at Mon, 14 May 90 20:37:40 EDT
expn hostmaster
250-Howie Kaye <howie@ivory.cc.columbia.edu>
250-Howie Kaye <\howie>
250-<jbaltz@cunixe>
250-Fuat C. Baran </f/sy/fuat/.backup/backup.mail>
250 Fuat C. Baran <\fuat>
quit
221 cunixf.cc.columbia.edu closing connection
Connection closed by foreign host.
%
```

We then try *finger*'ing again.

```
% finger fuat@cunixf.cc.columbia.edu
[cunixf.cc.columbia.edu]
fuat   Fuat C. Baran    Last login May 13 12:58 from ttyqd (sparky.cc.columb)
Project: CUCCA UNIX Systems Group and postmaster
Mail forwarded to \fuat, /f/sy/fuat/.backup/backup.mail

I am a member of the CUCCA UNIX Systems Group and CUCCA postmaster.  I
am one of the authors of Columbia-MM.

Electronic mail is the best way to get in touch with me.

For fastest results, send electronic mail related queries to 'postmaster',
general questions to 'consultant', mm problems to 'bug-mm' please.
In an emergency call the Operations Shift Supervisor at 854-2652 and have
him page someone from the UNIX Systems Group.

Passwords -- ''Use them like a toothbrush.  Change them often and don't
               share them with friends.''
                                        --Clifford Stoll
----
Internet: fuat@columbia.edu        U.S. MAIL: Columbia University
  BITNET: fuat@cunixf                         Center for Computing Activities
    UUCP: ...!rutgers!columbia!cunixf!fuat    712 Watson Labs, 612 W115th St.
   Phone: (212) 854-5128  Fax: (212) 662-6442 New York, NY 10025
%
```

Here we finally find an emergency phone number. If none of the above had yielded a phone number, we could also try *finger*'ing common usernames such as *root, operator, operate, system*, etc. and hopefully one of them would have a finger plan. If all else fails, we can try calling directory assistance for the city.

A.1. Is Your Information Available?

When dealing with the incidents mentioned above, we had the need to contact system administrators at other sites. In some cases we followed all of the preceding steps and still had trouble reaching somebody. It is a good idea to periodically check places such as the *whois* database, the SOA record in the DNS, and finger plans of system IDs and administrators to make sure that someone outside your site can reach you in an emergency with minimal effort. It is also a good idea to make sure that your local phone company has a general information number for your institution listed, as well as verifying that your internal phone operator knows the number of the computer center.

When all else fails someone may have to resort to sending electronic mail, so make sure IDs such as *root* are forwarded to people who are in charge of the system. However, in some cases sending email is inadvisable, in the event that the message may be intercepted.

The USENIX Association

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

Telephone: 415 528-8649
Email: office@usenix.org

The USENIX Association

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

Computing Systems, published quarterly by the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the dues paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

Telephone: 415 528-8649
Email: office@usenix.org

USENIX Supporting Members

Aerospace Corporation
AT&T Information Systems
Digital Equipment Corporation
Frame Technology Corporation
mt Xinu
Open Software Foundation
Quality Micro Systems
Sun Microsystems, Inc.
Sybase, Inc.